Sparse matrix vector multiplication on FPGA-based platforms

by

Kevin R. Townsend

A preliminary report submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering (Computing and Networking Systems)

Iowa State University

Ames, Iowa

2014

Contents

List of Tables

List of Figures

ABSTRACT

Chapter1. INTRODUCTION

This preliminary report outlines a plan to double the current performance of sparse matrix vector multiplication on FPGA platforms.

People use SpMV in a variety of applications including information retrieval, text classification, scientific computing and image processing. Often the SpMV operations are iterative or repeatative and require a large amount of computation. Eiganvector estimation often uses iterative SpMV operations. For example, the pagerank algorithm uses SpMV for eiganvector estimation.

For the most part modern CPUs perform SpMV well. Some may even say FPGAs desirve no consideration when computing SpMV, or say that computing SpMV on FPGAs is solely academic and would only help to design better ASICs, CPUs and GPUs for computing SpMV. We disagree. If you have an application that uses repeatative SpMV operations on large matrices then FPGAs are exactly the chips you should be looking at. When the matrix and vector sizes become large, around 10 million values, performance drastically decreases. To address this issue most people turn to GPUs.

However, GPUs have an interesting characteristic. In order to achieve good performance GPUs expand the storage size of the matrix. FPGAs do the opposite and compress the size of the matrix. This means matrices with more than 400 million values perform badly or do not fit in the GPU's RAM.

So GPUs are stuck between a rock and a hard place **?**. The rock being CPUs that compute SpMV on matrices with less than 10 million values well. The hard place being FPGAs that compute SpMV on matrices with more than 400 million values well (or at least not as badly as CPUs and GPUs).

In the chapter 2 we describe the previous approaches to SpMV on CPUs, GPUs and FPGAs.

In chapters 3, 4, 5, 6 and 7 we discuss our optimizations for FPGAs. In chapters 8 and 9 we present our predicted results and discuss the timeline to achieve these results.

Chapter2. BACKGROUND

In its simplest form sparse matrix vector multiplication is the operation $y = Ax$, where A is an M by N, x is a vector of length M, and y is a vector of length M. As Equation 2.1 shows, matrix vector multiplication is a series of dot products.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} A_{11}x_1+A_{14}x_4+A_{17}x_7 \\ A_{25}x_5+A_{28}x_8 \\ A_{32}x_3+A_{33}x_3+A_{36}x_6+A_{37}x_7 \\ A_{41}x_1 + A_{45}x_5 \\ A_{53}x_3 + A_{54}x_4 + A_{57}x_7 + A_{58}x_8 \\ A_{62}x_2 + A_{65}x_5 \\ A_{72}x_2 + A_{73}x_3 + A_{76}x_6 + A_{78}x_8 \\ A_{83}x_3 + A_{84}x_4 + A_{85}x_5 + A_{86}x_6 \end{bmatrix} = \begin{bmatrix} A_{11} & 0 & 0 & A_{14} & 0 & 0 & A_{17} & 0 \\ 0 & 0 & 0 & 0 & A_{25} & 0 & 0 & A_{28} \\ 0 & A_{32} & A_{33} & 0 & 0 & A_{36} & A_{37} & 0 \\ A_{41} & 0 & 0 & 0 & A_{45} & 0 & 0 & 0 \\ 0 & 0 & A_{53} & A_{54} & 0 & 0 & A_{57} & A_{58} \\ 0 & A_{62} & 0 & 0 & A_{65} & 0 & 0 & 0 \\ 0 & A_{72} & A_{73} & 0 & 0 & A_{76} & 0 & A_{78} \\ 0 & 0 & A_{83} & A_{84} & A_{85} & A_{86} & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} \tag{2.1}$$

Sparse matrices differ from dense matrices in that they contain mostly (usually more than 99%) zeros. For example, consider the matrix representation of the Facebook friends graph. Each row contains non-zero values representing friend connections and zero values representing non-friends. Being friends with .1% of Facebook users would require being friends with 10 million people, an impressive feat. In other words, from the time you started reading this paper 1000 people have joined facebook and are not friends with you. The average user has 100 friends. For this reason sparsity is usually measured in elements per row rather than a percent. The percent sparsity of the matrix keeps growing but the number of non-zero elements per row stays about constant.

2.1 Coordinate Formate (COO)

Dense matrices can be stored as an array of values. However if sparse matrices were stored this way they would require orders of magnitude more space than a simple alternative. The alternative, coordinate formate (COO), stores 3 arrays: a row index array, a column index array, and a value array. Coordinate formate (COO) is the simplest sparse matrix storage scheme. Every non-zero element is stored in 3 parts: a row index, a column index, and the element's value. For simplicity, this paper only concerns itself with double precision values. Using the example matrix, the COO format would be:

ROW: 0, 0, 0, 1, 1, 2, 2, 2, 2, 3, . . . 7, 7, 7

COLUMN: 0, 3, 6, 4, 7, 1, 2, 5, . . . 2, 3, 4, 5

VALUE: A_{11}, A_{14}, A_{17}, A_{25}, A_{28}, A_{32}, $\ldots A_{86}$

You will notice that the elements are traversed in row-major form. Row-major traversal starts at the left most element of the first row (A_{11}). Then proceeds to the next element to right (A_{14}). After arriving at the last element of a row the next element would be the left most element in the row below it (A_{25}). This is a convenient, but not necessary traversal of the matrix.

Calculating SpMV with this matrix format is straight forward and much faster than if the whole matrix was used. SpMV takes nnz multiplications and $nnz - M$ additions, where nnz is the number of non-zero values in the matrix and M is the height of the matrix. This totals $2 \times nnz - M$ floating point operations. However, the convention in the field uses a slightly incorrect but simpler $2 \times nnz$ to report performance, which we use to report our performance. The difference is usually only a slight over estimate of the actual performance, but the difference could be big if nnz/M is small.

2.2 CPU Processors

The sparsity of the matrix causes CPUs to perform below their potential. For example Intel publishes an average performance of 50 GFLOPS for matrix vector multiplication but publishes an average performance of 10 GFLOPS for SpMV.

To understand this look at the equation 2.1 again and count the number of times each value is accessed. The values in the matrix only get accessed once and the values in the vector only get accessed a couple times. This remains the same for large matrices, because, as mentioned, the number of non-zero values per row (nnz/M) often grows slowly for larger matrices. This means the computations to memory operations ratio is low. Compare this to matrix-matrix multiplication where the ratio is high and the CPU can perform at 1 TFLOP, almost the limit of the CPU.

The effect of this small ratio effects the CPU less when everything can fit in cache. Although it still exists, because L3 cache has some latency.

If you are impatient feel free to skip ahead to the section about SpMV on the FPGA. We would like to cover CPU and GPU optimizations first. We cover techniques and how they could apply to FPGA implementations.

2.3 Compressed Sparse Row Formate (CSR)

The first optimization is the simplest. The optimization compresses the row indices. When the elements are traversed in row major form the row indices change little. So instead of storing row indices the traversal index of the first element of each row is stored instead. The traversal index equals the number of non-zero elements that are traversed before the current element is reached. We use the term traversal index to prevent confusion when mentioning row and column index. Assuming the indices are stored as ints (4 bytes) and values as double (8 byte floating point values) this change saves up to 4 bytes per element or 25% of the total matrix storage size.

Compressed sparse row or CSR is a common compression scheme. It relys on the mentioned row-major traversal. The column and value arrays are the same as COO. A compressed row array replaces the row array. The row array usually doesn't change from one element to the next and when it does it only changes by increasing the index by one. This new compressed row array only marks when the row index is increased by one. The compressed row array for the example is shown:

COMPRESSED ROW ARRAY: 3, 5, 9, 11, 15, 17, 21, 25

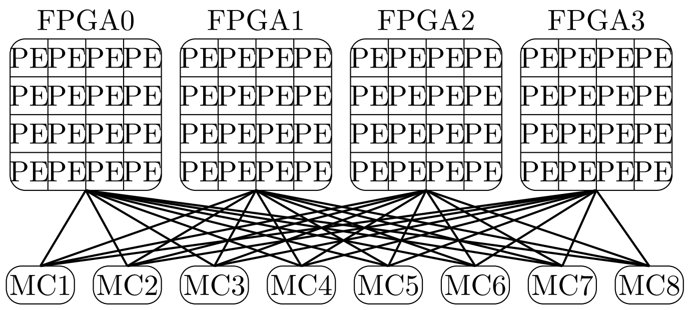


Figure 2.1: R^3 implementation on the Convey HC-2 coprocessor: 4 Virtex-5 LX330 FPGAs tiled with 16 R^3 SpMV processing elements each. Each Virtex-5 chip connects to all 8 memory controllers, which enables each chip to have access to all of the coprocessor's memory.

Table 2.1: Matrix Statistics

Matrix	**Field**	**dimensions**	**nnz**	**nnz/row**
dense	Example	2,000x2,000	4,000,000	2,000
consph	FEM/Speres	83,334x83,334	6,010,480	72
cant	FEM/Cantilever	62,451x62,451	4,007,383	64
rma10	FEM/Harbor	46,835x46,835	2,329,092	49
qcd5_4	QCD	49,152x49,152	1,916,928	39
shipsec1	FEM/ship	140,874x140,874	3,568,176	25
mac_econ_fwd500	Economics	206,500x206,500	1,273,389	6.2
mc2depi	Epidemiology	525,825x525,825	2,100,225	4.0
scircuit	Circuit	170,998x170,998	958,936	5.6

Table 2.2: Matrix Statistics

Matrix	**R^3 Gflops**	**2× Intel E5-2690**	**Nvidia Tesla M2090**
dense	13.6	14	**23**
consph	8.7	11	**15**
cant	12.7	12	**17**
rma10	13.6	**24**	11
qcd5_4	12.8	**30**	20
shipsec1	7.9	10	**11**
mac_econ_fwd500	5.9	**23**	6
mc2depi	6.2	21	**22**
scircuit	6.2	**12**	6

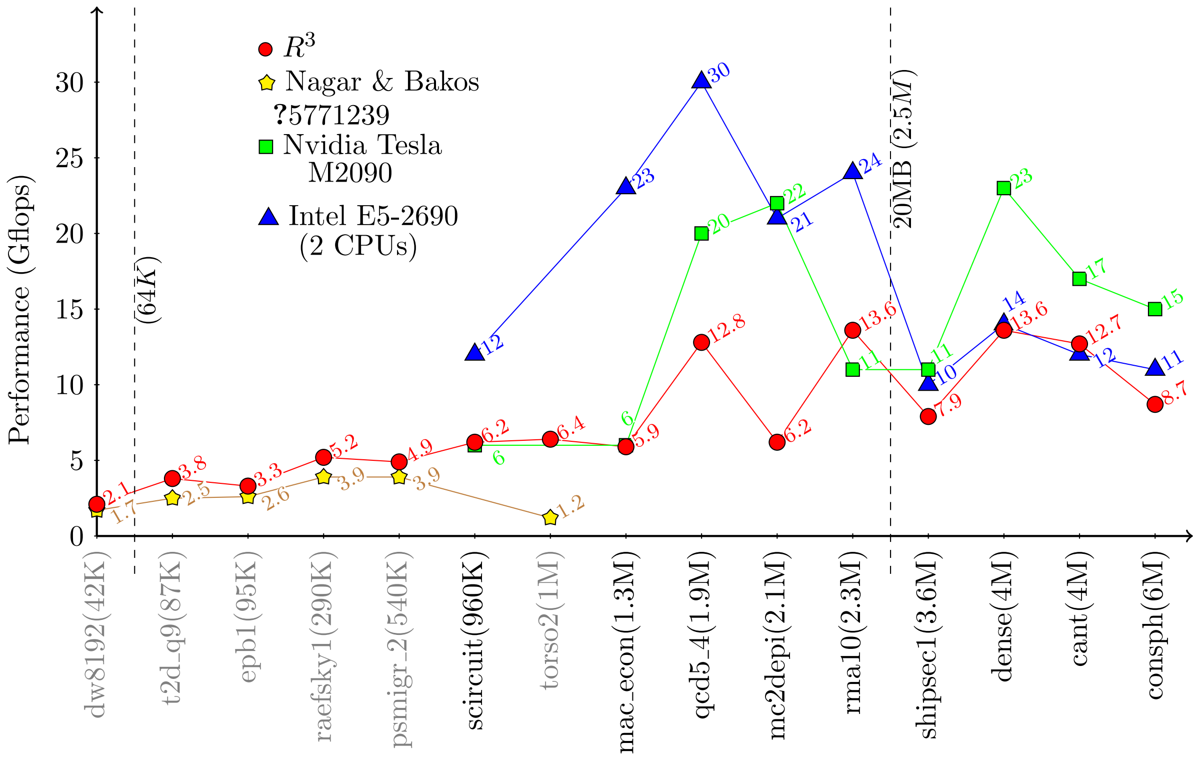


Figure 2.2: *nnz* vs Performance on each platform. The small matrices, ones around 64K or less, performed poorly on most platforms, due the overhead. CPUs experience the opposite effect. They take a performance hit once the matrix no longer fits in cache.

2.4 Block Sparse Row Format (BSR)

Compression schemes often take advantage of the clumpy structures of sparse matrices. Blocking or register blocking stores dense sub blocks of the matrix together. This again reduces the matrix storage size by storing fewer indices. Some explicit zeros are added to complete the subblocks.

The block sparse row (BSR) storage format is one such block storage scheme. It stores the row and column indices of the top left of the block and stores the values of the block in row major form. This matrix format stores the matrix as the some of 2 matrices one in BSR format the other in CSR or COO.

2.5 Delta Compression

Another index compression scheme, delta compression, compresses indices aggressively by storing delta distances. A delta is the distance between the previous and current matrix element. The average number of bits is quite small (discussed in chapter 5). The paper is vage on the details, however since this uses variable length encoding some encoding scheme is required. This saves a lot of space, but the results are medicore. The time to decode the deltas into row and column indices requires a non-trivial amount of processing time that potentially could be used for floating point operations. However, FPGAs can dedicate space for decoding, but we will get to that later.

2.6 Value Compression

The same paper uses value compression, for the matrix values. Again the details are a little vague, but the idea was to take advantage of the fact values repeat. Eventhough this saves a lot of space for some matrices the results are again mediocore.

CPU optimiziations also include changing the matrix traversal for better vector reuse. BSR does this to a small extent. One method called Cache blocking traverses large reletively dense subblocks individually before proceeding to the next block. The dimensions of the block are around the size of available cache. This method has similarities to our row column row (RCR) traversal.

2.7 GPU

GPUs play the computation game differently the CPUs. To show this let us compare a high end CPU (Intel Xeon E5-9999) and a high-end GPU (Nvidia Tesla 9999). The GPU has a max throughput of 1TFLOPS (double precision). The CPU has a max throughput of 200GFLOPS (double precision). The GPU has 1MB of cache (although a texture cache is available). The CPU has 20MB of cache. The cache is growing as well. The previous testla () hat 2MB of cache. THe previous Xeon had 15MB of cache. The GPU is a vector machine making it hard to get good performance on unstructured computation.

To enable better performance **?** used a storage formate designed for vector processors. ELLPACK stores the same number of values for each row. Rows with fewer values than the row with the most values are padded with zeros.

The ELLPACK formate for the example would be:

COLUMN: 0, 3, 6, 0, 4, 7, 0, 0, 1, 2, 5, $\ldots$ 2, 3, 4, 5

VALUE: A_{11}, A_{14}, A_{17}, 0, A_{25}, A_{28}, 0, 0, A_{32}, $\ldots A_{86}$

The authors also deal with abnormally large rows by computing the values in COO format. Simillarly to BSR this stores the matrix into the sum of 2 matrices.

2.8 FPGA

Like GPUs FPGAs play by their own computation rules. Although FPGAs usually do not have an advertised FLOPS perforamance one can be calculated by creating a matrix matrix multiplication engine to load on the FPGA.

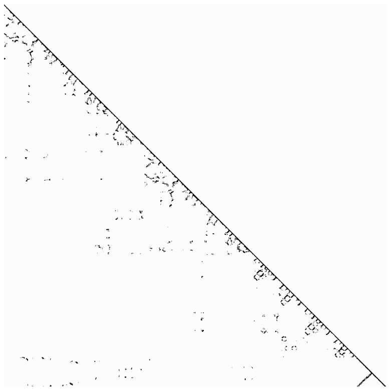
The basic idea of an FPGA is to design the processor you want and that design can be loaded on to an FPGA

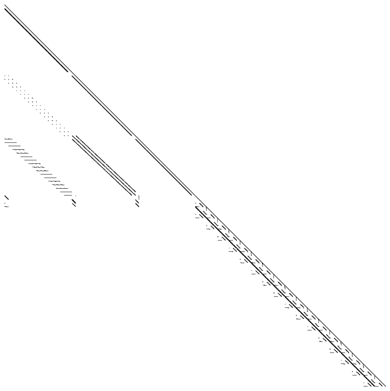
Achieving the computation capacity to be competitive with CPUs and GPUs for SpMV

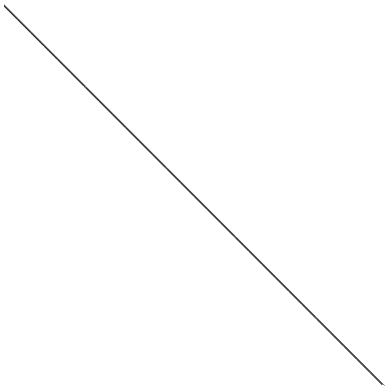
2.9 Reconfigurable Computing Niche

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. The next chapter describes our vision of how FPGAs will outperform other processors when it comes to SpMV.

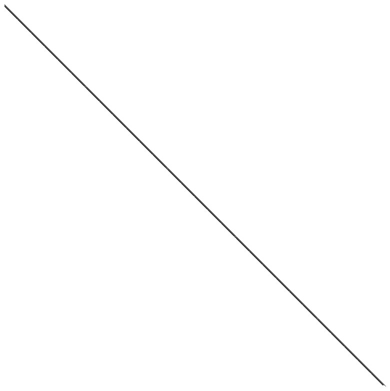
(a) dense

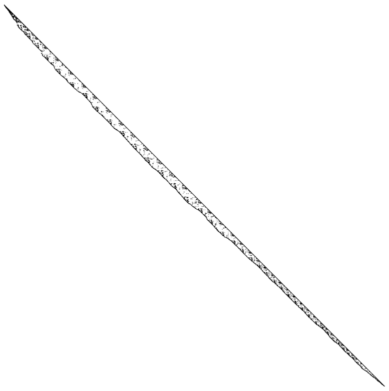

(b) pdb1HYS

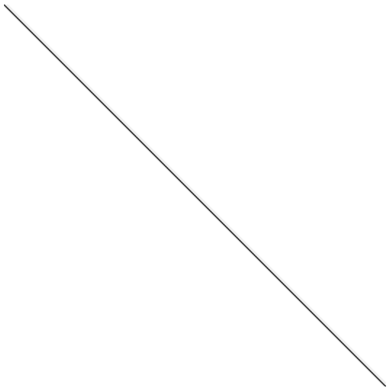

(c) consph

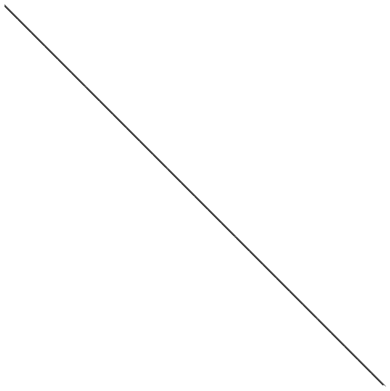

(d) cant

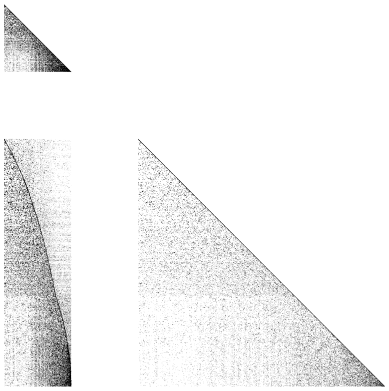
(e) pwtk

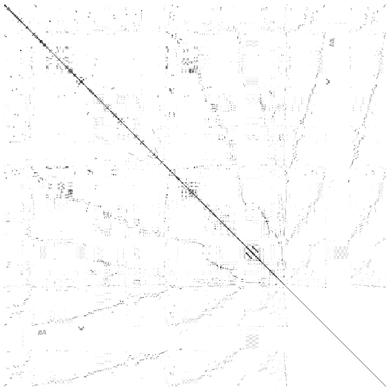

(f) cant


(g) shipsec1


(h) mac_econ_fwd500


(i) cant


(j) cop20k_A


(k) scircuit

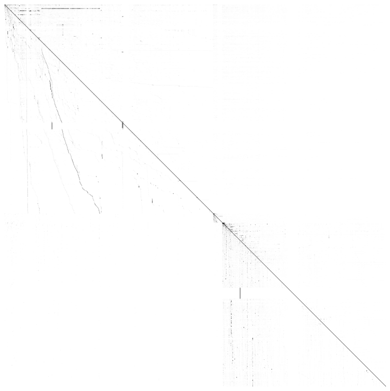

(l) webbase-1M

Figure 2.3: The density plots of the matrices used for testing

Chapter3. SpMV on FPGA METHODOLOGY

In the previous chapter we have discussed how others have approached computing on FPGAs and other processors. We borrow apon the good ideas and add our own.

Our design methodology consists of 3 design pillars. The first pillar is designing a multiply accumulator that does not stall and maintains multiple intermediate values. The second pillar is designing a matrix traversal that enables reuse of vector values and improves matrix compression. The third pillar is designing a matrix compression scheme with a high compression ratio and has a hardware ameanable decoder.

These pillars rely on each other. The first pillar, the multiply accumulator, has to accumulate multiple rows at a time to allow different traversals, the second pillar. The multiply accumulator should also not (or rarely) stall. A multiply accumulator that stalls regularly can not maintain high throughput.

The second pillar, matrix traversal, primarily helps with vector reuse. Column traversal has a major effect on vector reuse. Many people argue that vector caching is the best way to achieve vector reuse for FPGAs. We disagree. With the ability to use column traversal in a horizontal subsection of say 1000 rows one can perfectly reuse vector values in this section. This requires the storage of 1000 intermediate y values or 8KB. Compare this to caching. Assume there are 10 non-zero elements per row and assume each vector value gets accessed twice. Then to achieve good caching the cache must support 5000 values or 40KB. This also ignores storing the vector indices of the cached values. So, in this example, storing intermediate values is more than 5 times more space efficient than vector caching.

The second advantage of mixing row and column traversal is that it leads to smaller deltas. In this paper a delta is the traversal distance between a matrix element and its preceeding matrix element in the traversal.

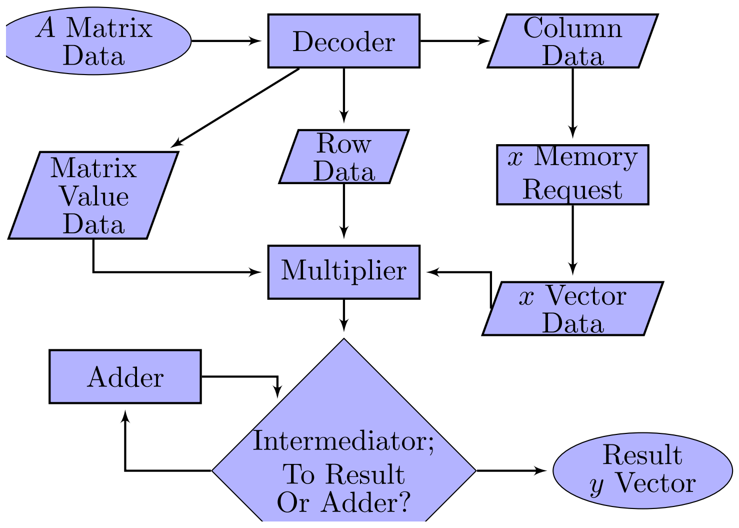


Figure 3.1: Data flow of an R^3 SpMV processing element. The processing element needs the column data before accessing the vector data.

The third pillar may be the most important for FPGAs. Compression of the matrix has a large amount of importance, because reading the matrix takes up a majority of the memory bandwidth. Currently the SpMV field views not counting preprocessing of the matrix towards the SpMV runtime allowable. This is because SyMV usually used for iterative and repeatative methods. We agree with this sentiment.

Using deltas to compress indices is the first and easiest step towards this pillar. Many compression implementations try to align variable length encoding to 4 bit or other size boundries. We give little regard to boundries because we find the added compression to be worth the extra FPGA space the decoder needs.

Value compression is tricky but has a potential to save large amounts of space and thus memory bandwith. Values repeat more than one would expect in matrices. Taking advantage of this repeatition is the biggest step towards good compression. Figure 3.3 shows hub much of an effect this pattern has on the performance of our previous SpMV implementation R^3.

When these pillars are in place the dataflow of the design still looks similar to other implementations (figure x). The architecture diagram (Figure x) also follows the same general fol of the dataflow diagram.

We seperate our current and planned contributions into 5 pieces, the next 5 chapters.

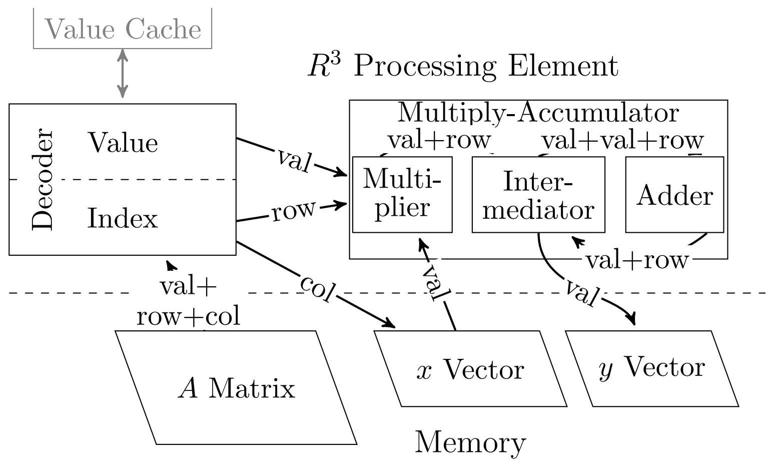


Figure 3.2: A single R^3 processing element. The arrows show the flow of data through the processing element. Although this diagram shows the memory access to each of the 3 places in memory as separate, they share one memory port. The diagram also does not show the FIFOs that help keep the pipeline full.

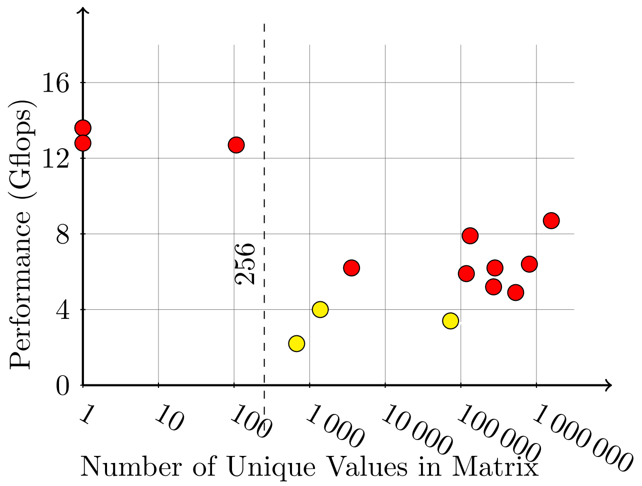


Figure 3.3: Unique values in a matrix vs the performance of R^3. Matrices with fewer than 256 unique values (only common elements exist) enables R^3 format to compress much better. The ●'s are outliers due to their size (see Figure 2.2).

Chapter 3 focuses entirely on the first pillar, the multiply accumulator design. Chapter 4 focuses on pillars 2 and 3 with disscussing matrix traversal and compressing indices with delta compression. Chapter 5 focuses focuses on the second half of pillar 3, value compression. We found good value compression requires a large amount of memory therefore Chapter 6 focuses on the design of a large memory shared by multiple processing elements.

Chapter4. MULTIPLY-ACCUMULATOR

A high throughput 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 **?**Goldberg:1991:CSK:103162.103163.

In R^3 we designed a block called an Intermediator (Section 4.0.1) capable of storing 32 intermediate y vector values. In our next design we indend to expand this to 1024 (the depth of one dual port BlockRAM in most Xilinx chips). Both designs have an interesting side effect that the allow the matrix to be traversed in a loosely row major traversal and the MAC will still work correctly. The step to from 32 to 1024 intermediate values allows more freedom in

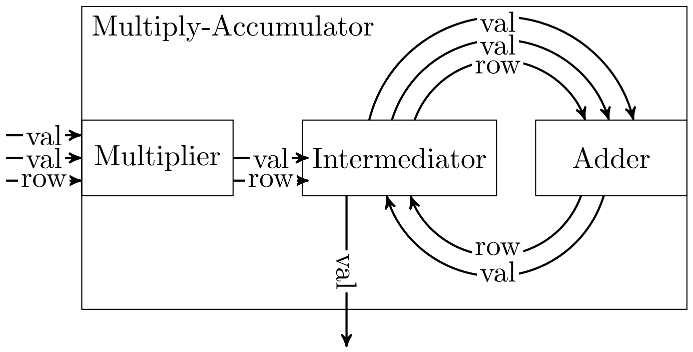


Figure 4.1: The no-stall multiply-accumulator block handles multiple intermediate values at a time. This allows multiple intermediate values in the adder pipeline.

the traversal. The rest of the chapter discusses the new design. The matrix elements in one set of 512 rows can be traversed in any way just as long as all the elements are traversed before going to the next 512 rows. Later in chapter 5 we discuss traversals that abide by this rule and allow for easy reuse of x vector values. We plan to use a traversal we called row column row (RCR) traversal.

4.0.1 Intermediator

We call the component that handles the intermediate y values the intermediator. The Intermediator (Figure 4.2) 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 4.2) 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. In R^3 we approach this problem by limiting the number of active intermediate values to 32. In our new design we will use a distributed RAM with width of 1 bit to keep track of the state of each slot. Since distributed RAM only has 1 write port we need to do something clever. The first option would be to double the frequency of the distributed RAM to achieve two writes every clock cycle. The second option is to create a dual port RAM with Banking. Banking is disscussed in chapter 9.

The memory has four states with we call the red, yellow, green and white states. The memory is partitioned into 2 parts an upper and lower part. Once the accumulation starts one of these parts will be in the red active state. Once the incoming values move to the next 512 row section of the matrix the active state transitions to the yellow fading state, and the other half of the memory is now in the active state. Recall our traversal rule is that each 512 rows must be traversed before proceeding to the next 512 rows. The yellow fading state exists because values are still be accumulated in the previously active memory. The memory will always be accumulated in 80 clock cycles. At that point the faded state transitions to the green "read to store" state. Once the values have been sent out to be stored the memory transitions to the white idle state.

To understand why 80 cycle cycles are needed to ensure the accumulation has finished after no new values arrive from the multiplier, 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.

Many cases occur when accumulating values in multiple rows and the Intermediator handles each case properly:

Case 1: (Figure 4.2g) 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.

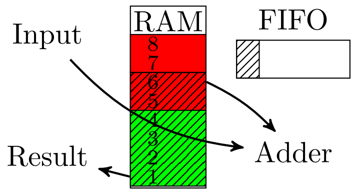
Case 2: (Figure 4.2d) Only one value arrives (valid) and the row corresponds to an empty cell. The value goes into the empty cell. If the "to result" window has values, it outputs a result, and if the FIFO has values it outputs a set to the adder.

Case 3: (Figure 4.2a) 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.

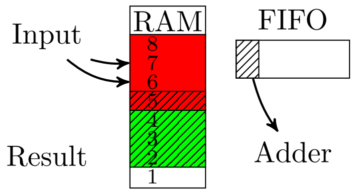
Case 4: (Figure 4.2b, 4.2h) Both values have row indexes that correspond to empty cells in the Block RAM. Both values get stored in the Block RAM and both cells switch to full. If the FIFO from the Block RAM to the output has values it sends one set of values to the output.

Case 5: (Figure 4.2f) 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.

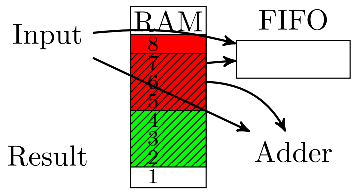
Case 6: (Figure 4.2c) Both values have row indexes that correspond to full cells in the Block RAM. One input value and corresponding Block RAM cell goes to the output. The output can



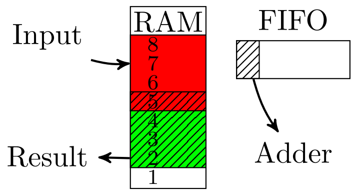
(a) First clock cycle, 1 pair of values get sent to the adder.



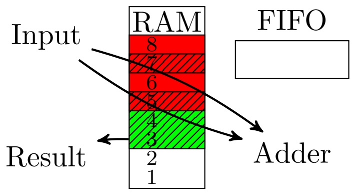
(b) Second clock cycle, 2 element gets stored in RAM.



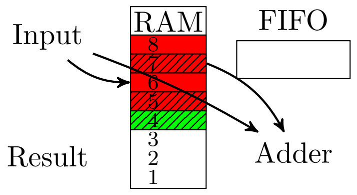
(c) Third clock cycle, the 2 inputs correspond to full cells in the Block RAM.



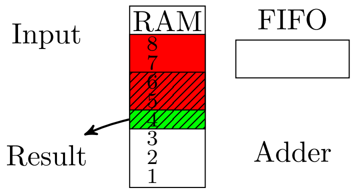
(d) Fourth clock cycle, 1 element gets stored in RAM.



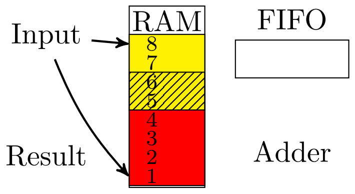
(e) Fifth clock cycle, the row indexes of the 2 inputs equal each other.



(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 4.2: This shows a simple example of the Intermediator running for 8 clock cycles. For demonstration, the size of the RAM is 8 instead of 1024.

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 4.2e) 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.

To help explain, consider a simpler case where the depth of the intermediator is 8 instead of 1024. Figure 4.2 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 values would occur in the result. Again we can do a worse case analysis. Wo assume each row has at leas one value. In our opinion preprocessing matrices to either remove empty rows or add explicit zeros is fair. Our worst case would be if all teh rows only had 1 value. This means the active memory would be active for 512 cycles but storing the previous values would take 512+80 cycles. However the worst case stall would decrease throughput by 80/512 (10%). This worst case behavior does not occur in practice.

Chapter5. MATRIX COMPRESSION

Not much work focuses solely on matrix compression. However, matrix compression for SpMV bas been studied. Most approaches split the problem into matrix index compression and value compression. We agree with this approach.

We discuss floating point value compression in the next chapter. In this chapter we discuss index compression, but first we want to discuss matrix traversal.

5.1 Matrix Traversal

We show matrix traversal and index compression are linked. We use deltas to compress indices. In R^3 we use a traversal called global row major local column major. For the rest of teh paper we call this traversal column row traversal.

5.1.1 GRMLCM

5.1.2 Column Row Traversal

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. We call this column row traversal. 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, with column height equal to four the traversal would read as: $[A_{11}, A_{41}, A_{32}, A_{33}, A_{14} \ldots]$.

Table 5.1: General information on test matrices and performance of previous compression methods

Matrix	**COO (bytes/nnz)**	**CSR (bytes/nnz)**	**COO+gzip**	**CSR+gzip**	R^3 **Format**
dense2[a]	16.00	12.06	0.85	0.51	2.27
pdb1HYS	16.00	12.15	7.55	6.68	9.60
consph	16.00	12.65	3.23	2.21	7.60
cant	16.00	12.03	4.39	4.32	9.13
pwtk	16.00	12.10	0.55	0.32	2.58
rma10[a]	16.00	12.71	4.67	3.56	5.82
qcd5_4[a]	16.00	12.06	5.43	5.42	9.11
shipsec1	16.00	12.00	0.06	0.04	2.26
mac_econ_fwd500	16.00	13.00	4.09	3.01	5.46
mc2depi	16.00	12.08	4.57	4.46	8.81
cop20k_A	16.00	12.08	0.35	0.21	2.46
scircuit	16.00	12.16	3.68	2.92	7.19
webbase-1M	16.00	13.29	2.18	1.73	2.64
average[b]	16.00	12.34	3.20	2.72	5.76

[a] Boolean matrices
[b] Excludes boolean matrices (TODO)

At this point we have a traversal that abides by the rules needed for the multiply accumulator and reuses vector values. However, no thought has yet been given to compression. To better understand compression we analyze several compression schemes.

5.1.2.1 Delta Compression

?4625888 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).

Many papers use delta compression **?**smac:r3, smac:kourtis, smac:kestur. Delta Compression stores the distance between the previous and current element. This results in smaller values that require fewer bits. The overhead of encoding these bit length varies among the different

Table 5.2: Detailed analysis of index compression and performance of Smac

Matrix	**COO**	**CSR**	**CSR.gz**	**Delta bits**	**GRMLCM16**	**GRMLCM256**	**GRMLCM1024**	**Smac**
dense2	8.00	4.00	0.03	0.00	0.00	0.00	0.00	0.13
pdb1HYS	8.00	4.03	0.14	0.06	0.05	0.05	0.06	0.21
consph	8.00	4.06	0.19	0.13	0.10	0.11	0.12	0.30
cant	8.00	4.06	0.40	0.12	0.10	0.11	0.11	0.31
pwtk	8.00	4.08	0.17	0.09	0.05	0.06	0.07	0.22
rma10	8.00	4.08	0.20	0.11	0.08	0.09	0.10	0.28
qcd5_4	8.00	4.10	0.31	0.24	0.16	0.20	0.22	0.46
shipsec1	8.00	4.16	0.86	0.41	0.27	0.34	0.37	0.72
mac_econ_fwd500	8.00	4.65	1.48	0.77	0.56	0.61	0.64	1.16
mc2depi	8.00	5.00	1.78	1.11	0.50	0.81	0.88	1.72
cop20k_A	8.00	4.15	1.07	0.58	0.42	0.49	0.53	0.90
scircuit	8.00	4.71	1.61	0.85	0.48	0.58	0.66	1.10
webbase-1M	8.00	5.29	1.35	1.57	0.46	0.55	0.64	1.22
average[a]	8.00	4.36	0.80	0.50	0.27	0.33	0.37	0.72

[a] Excludes dense matrix

schemes. The storage size per element of these delta values using only the bits necessary are in column 5 of table 5.2 (this does not include overhead).

We also choose to use the general compression program gzip in the comparison as well. Again table 5.2 shows the compression of gzip on top of the CSR format. Gzip does very well, in one case (webbase-1M) even takes less space than storing only delta bits. This is particularly surprising considering that extra overhead is needed to decode these delta bits. The reason this occurs is because large delta values can represent a short vertical jump. gzip remembers previous column indexes and therefore can compress them easily.

It seems hard to believe that gzip would be the best compression scheme. However, we notice column row traversal has smaller deltas than row major traversal. The table 5.3 shows this distribution. This is because column row traversal does make vertical steps.

Row column traversal does improve the index compression for all matrices and a significant improvement for some. However it is disappointing to see that larger column heights lead to worse performance. To keep the larger column heights for better vector reuse, but still achieve small deltas we propose short row traversal in the column traversal. In other words row column row (RCR) traversal. The row width will be experimentally choosen, but 16 seems like a good guess. This means 16 vector values instead of 1 need to be cached, but this seems achievable.

To illustrate let us look at the traversal in the example matrix. In this example we set the row and column parameters to 2 and 4 respectfully. In this case the RCR traversal is: $[A_{11}, A_{32}, A_{41}, A_{14}, A_{33} \ldots]$

However it does not fix the issue that extra overhead is needed to decode the delta bits. We created our own encoding scheme. The rest of this section describes our encoding scheme. To reduce the overhead we use a variable sized encoding scheme. We looked at the distribution of bit lengths over all the matrices in the test cases.

5.1.3 Smac index compression

The easiest trend to see from the distribution table is that more than half of the elements usually occur immediately after another element. So we choose to simply encode these as a

Table 5.3: The distribution of the bit lengths required to store the delta length when using GRMLCM16 traversal

Matrix	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**9+**
dense2	%100.00	%0.00	%0.00	%0.00	%0.00	%0.00	%0.00	%0.00	%0.00	%0.00	%0.00
pdb1HYS	%89.23	%0.44	%2.39	%2.43	%4.77	%0.01	%0.19	%0.09	%0.13	%0.05	%0.26
consph	%80.47	%1.58	%0.03	%3.30	%12.89	%0.71	%0.02	%0.00	%0.00	%0.48	%0.52
cant	%75.74	%5.43	%0.08	%2.99	%14.34	%0.58	%0.40	%0.12	%0.02	%0.01	%0.29
pwtk	%87.88	%1.49	%1.25	%3.07	%5.87	%0.04	%0.01	%0.00	%0.00	%0.01	%0.38
rma10	%81.63	%0.52	%4.43	%4.17	%7.66	%0.15	%0.56	%0.18	%0.07	%0.09	%0.53
qcd5_4	%67.63	%0.11	%3.85	%7.10	%17.36	%2.62	%0.00	%0.21	%0.00	%0.00	%1.12
shipsec1	%40.79	%11.53	%7.76	%3.74	%23.43	%9.51	%0.40	%0.45	%0.24	%0.36	%1.81
mac_econ_fwd500	%16.16	%3.98	%11.35	%7.30	%15.28	%12.42	%9.75	%6.41	%6.05	%3.77	%7.52
mc2depi	%23.36	%0.00	%0.00	%0.01	%24.92	%47.00	%0.02	%0.00	%0.00	%0.01	%4.68
cop20k_A	%50.18	%2.14	%1.55	%1.83	%24.16	%2.65	%1.11	%1.10	%1.07	%0.95	%13.26
scircuit	%37.71	%3.00	%2.71	%2.62	%25.65	%8.14	%3.45	%2.71	%2.27	%1.51	%10.24
webbase-1M	%46.55	%2.40	%1.70	%1.27	%5.57	%30.67	%0.74	%0.50	%0.36	%0.25	%9.99
average[a]	%58.11	%2.72	%3.09	%3.32	%15.16	%9.54	%1.39	%0.98	%0.85	%0.62	%4.22

[a] Excludes dense matrix

1_2. Now that we use 1_2 as a code, in order for the codes to be uniquely decoded, all the other codes must have a leading 0 (eg 110_2, 00_2).

Our second observation is that bit length groups can be combined at little cost. For example 10 bit deltas and an 11 bit delta can both be encoded as an 11 bit delta. This wastes 1 bit of the 10 bit delta encoded as 11 bits. So grouping by 2s would waste an average of 0.5 bits. Then grouping by 3s wastes 1 bit per element. Groups of 4 wastes 1.5 bits ect. So there is a trade off between using more codes, therefore longer codes, and wasting bits to do groupings. Groups of 3 seemed to work nicely.

Our third observation was that longer delta lengths generally occur less frequently. The frequency is similar to a exponential decreasing function. So we made the codes larger for the larger deltas.

In the end our encoding was the following: 0 bits : 1_2, 1-3 bits : 10_2, 4-6 bits : 100_2, 7-9 bits : 1000_2 ect. In other words: a "1" followed by N 0s, where N is the N^{th} group of 3 delta lengths. The last column of 5.2 shows the performance of this scheme.

Table 5.4: Detailed value compression analysis and performance comparison

Matrix	**uncompressed**	**Unique Values**	**Unique/nnz** $\times 8$	**256 Common**	**GZIP**	**FPC**	**Smac Unlimited Compression**	**Smac 4K Compression**
dense2[a]	8.00	1.00	0.00	1.00	0.01	0.50	0.75	0.76
pdb1HYS	8.00	1.10×10^6	4.08	7.99	4.15	7.99	5.07	7.91
consph	8.00	1.24×10^6	3.28	7.99	5.10	7.95	4.77	7.91
cant	8.00	1.07×10^2	0.00	1.00	0.11	0.91	0.89	0.90
pwtk	8.00	3.63×10^6	5.04	7.95	4.29	7.37	5.83	7.81
rma10[a]	8.00	1.00	0.00	1.00	0.01	0.50	0.75	0.76
qcd5_4[a]	8.00	1.00	0.00	1.00	0.01	0.50	0.75	0.77
shipsec1	8.00	8.86×10^4	0.56	6.39	2.08	3.80	2.34	4.06
mac_econ_fwd500	8.00	1.08×10^5	1.36	5.20	0.73	1.45	2.32	2.03
mc2depi	8.00	3.58×10^3	0.00	4.94	1.24	5.01	1.58	1.51
cop20k_A	8.00	9.56×10^5	5.84	7.97	5.53	7.97	5.92	7.85
scircuit	8.00	8.82×10^4	1.44	5.41	1.95	3.68	2.61	3.56
webbase-1M	8.00	5.65×10^2	0.00	1.48	0.38	1.92	1.10	1.11
average[b]	8.00	7.22×10^5	2.16	5.63	2.56	4.81	3.24	4.46

[a] Boolean matrices

[b] Excludes boolean matrices

Chapter6. FLOATING POINT COMPRESSION

Many datasets contain repeated values**?**floatZip:Kourtis,floatZip:Burtscher. Figure 6.1 shows an analysis of this. General compression schemes like gzip and algorithms specific to floating point compression like FPC do not single out this particular feature. However, we single out this feature for better compression.

Take the following simple scheme: store the repeated values separately form the rest of the data stream. With this scheme an index replaces each repeated value in the original data stream. This simple scheme works remarkably well for some datasets. However, we can achieve better compression. For example, when many repeating sequences of values exist, compressing them can vastly outperform the previous compression scheme. In total fzip takes advantage of three compressible features of datasets: repeating sequences (more than 8 bytes long), repeating values (8 bytes long) and repeating prefixes (less than 8 bytes long).

Floating point compression has multiple uses. In scientific computing floating point compression can improve the performance of parallel computing applications. Manycore computers have steadily been increasing computation capacity, but internal communication does not increase as fast. Compression has increased the performance of MPI implementations**?**floatZip:Ke. Also, floating point compression has been used to accelerate RAM access in applications like SpMV (Sparse Matrix Vector Multiplication)**?**floatZip:Kourtis,floatZip:Townsend.

6.1 Previous work

It was noted in **?**smac:kourtis that sparse matrices often have repeated values. This is the focus of our value compression. R^3 had a simple scheme using this. It stored the 256 most

Table 6.1: Detailed value compression analysis and performance comparison

Matrix	**uncompressed**	**Unique Values**	**Unique/nnz ×8**	**256 Common**	**GZIP**	**FPC**	**Smac Unlimited Compression**	**Smac 4K Compression**
dense2[a]	8.00	1.00	0.00	1.00	0.01	0.50	0.75	0.76
pdb1HYS	8.00	1.10×10^6	4.08	7.99	4.15	7.99	5.07	7.91
consph	8.00	1.24×10^6	3.28	7.99	5.10	7.95	4.77	7.91
cant	8.00	1.07×10^2	0.00	1.00	0.11	0.91	0.89	0.90
pwtk	8.00	3.63×10^6	5.04	7.95	4.29	7.37	5.83	7.81
rma10[a]	8.00	1.00	0.00	1.00	0.01	0.50	0.75	0.76
qcd5_4[a]	8.00	1.00	0.00	1.00	0.01	0.50	0.75	0.77
shipsec1	8.00	8.86×10^4	0.56	6.39	2.08	3.80	2.34	4.06
mac_econ_fwd500	8.00	1.08×10^5	1.36	5.20	0.73	1.45	2.32	2.03
mc2depi	8.00	3.58×10^3	0.00	4.94	1.24	5.01	1.58	1.51
cop20k_A	8.00	9.56×10^5	5.84	7.97	5.53	7.97	5.92	7.85
scircuit	8.00	8.82×10^4	1.44	5.41	1.95	3.68	2.61	3.56
webbase-1M	8.00	5.65×10^2	0.00	1.48	0.38	1.92	1.10	1.11
average[b]	8.00	7.22×10^5	2.16	5.63	2.56	4.81	3.24	4.46

[a] Boolean matrices

[b] Excludes boolean matrices

common values, so those common values could be represented as one byte. The performance of this scheme is shown in the column "256 common" in table 6.1.

Taking a step back, uncompressed data would take 8 bytes per element. Any compression scheme should take less than 8 bytes per element. We looked at **?**smac:burtscher describing it's own compression scheme FPC. FPC performs OK. This scheme looks for repeated patterns. However it does not exploit the fact most of its compression comes from exact (8 byte) value repeats.

Gzip performs quite well too. We have a general understanding of how Gzip works. We suspect the reason for the good performance is the large memory space and the use of looking up repeated 8-byte memory sequences.

Our focus on using repeated values is reinforced by looking at the number of unique values. If only the unique values were stored the average compression would be 2.16 bytes per element. This can not be used by itself since this ignores the indexing required to access these values, but this gives an estimate of the possible compression size.   In the remainder of this paper we talk about an analysis of floating point datasets (section 6.2), our approach to floating point compression (section 6.3) and our results (section **??**).

6.2 Floating-Point Value Analysis

Continuing the analysis from section **??**, figure 6.1 shows an analysis of the repeating values in each of the datasets used for testing. Several characteristics of this analysis suggest that compressing repeating values would perform well. For example, in half of the datasets at least 80% of the values repeat.

Another pattern exists among the prefixes of the values. To understand why, look at the floating point data structure. Double-precision floating-point values have 3 parts: a sign bit, 11 exponent bits and 52 fraction bits. Values close to each other often share the same sign (some datasets only contain positive numbers). Likewise close values also often share the most significant bits of the exponent. In fact the bits in floating-point values already exist in most likely shared to least likely shared sorted order: {sign bit, most significant exponent bits, least significant exponent bits, most significant fraction bits, least significant fraction bits}.

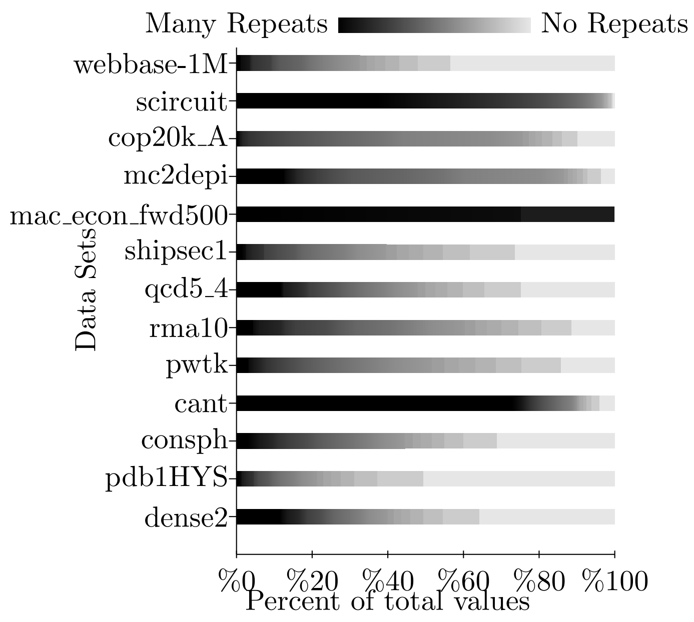


Figure 6.1: The above figure shows the distribution of repeats in each dataset. Each shade represents a different number of repeats. For instance: ■:> 512, ■:16, ■:2, ■:1(no repeats).

We gauged the strength of the pattern in a particular dataset by looking at how many prefix bits the adjacent values share. Figure 6.2 describes this analysis. From this figure we see that the first byte or so often repeats. However there usually exists a rapid decline in shared bits after this point.

Datasets might also have repeating patterns of values. For example the sequence, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0, has an obvious pattern. One can use the Burrows-Wheeler Transform**?**floatZip:Burrows to analyze these patterns. Figure **??** describes this algorithm some, however many other sources describe this algorithm in more detail**?**floatZip:Burrows,floatZip:Salomon. Figure 6.3 analyzes the number of repeats that appear after the Burrow-Wheeler Transform. As the figure shows 4 of the 13 test cases have a lot of patterns, but the rest have relatively few.

6.3 Our Approach

Our approach takes advantage features in section 6.2. The algorithm compresses with prefix and value compression.

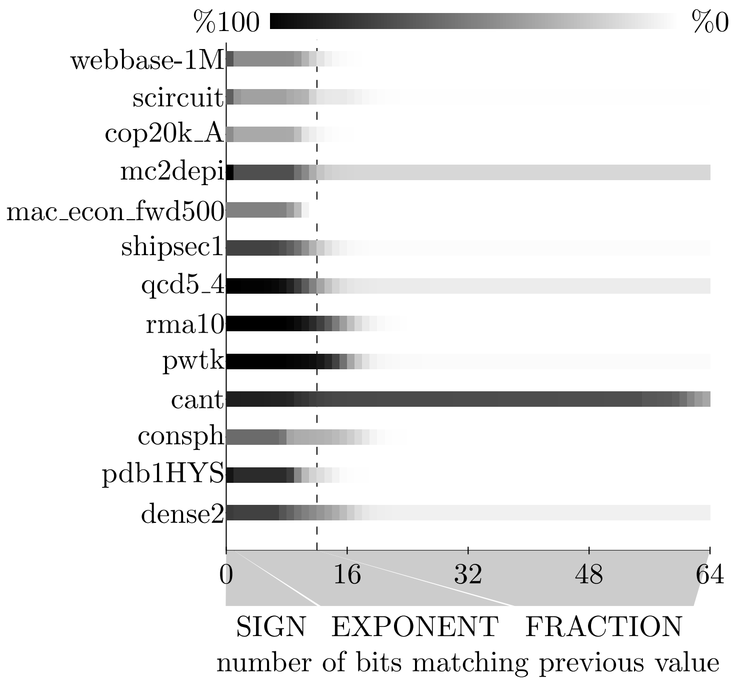


Figure 6.2: The above figure represents local prefix prediction. The figure shows the density function of 2 adjacent values sharing at least x number prefix bits. All of the data sets start at $(0, \%100)$. The curves end at the percent of values that are identical to their previous value for that dataset.

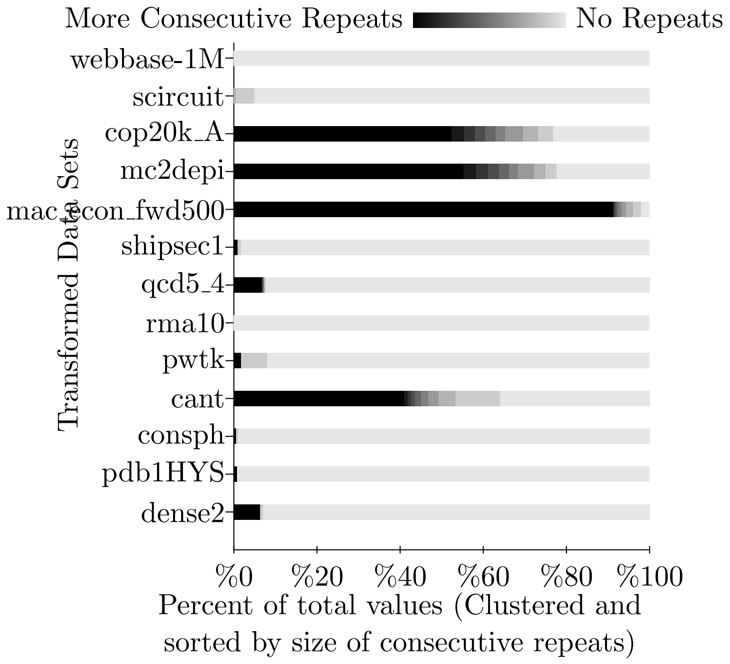


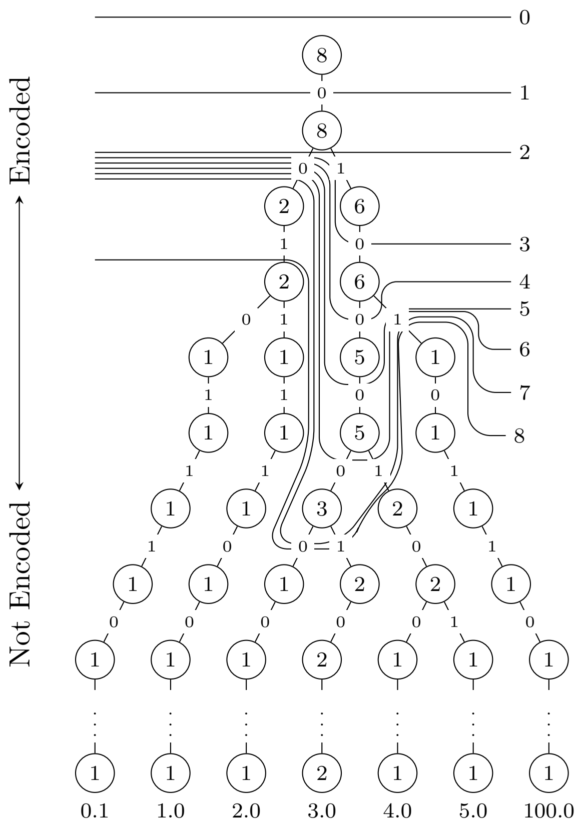
Figure 6.3: Pattern analysis using the Burrows-Wheeler Transform. Each shade represents the number of consecutive repeats in a repeating sequence. ■ represents sequences longer than 9. ▩ represents sequences of length 5. □ represents sequences equal to 1 (non-repeating).

6.3.1 Prefix Compression

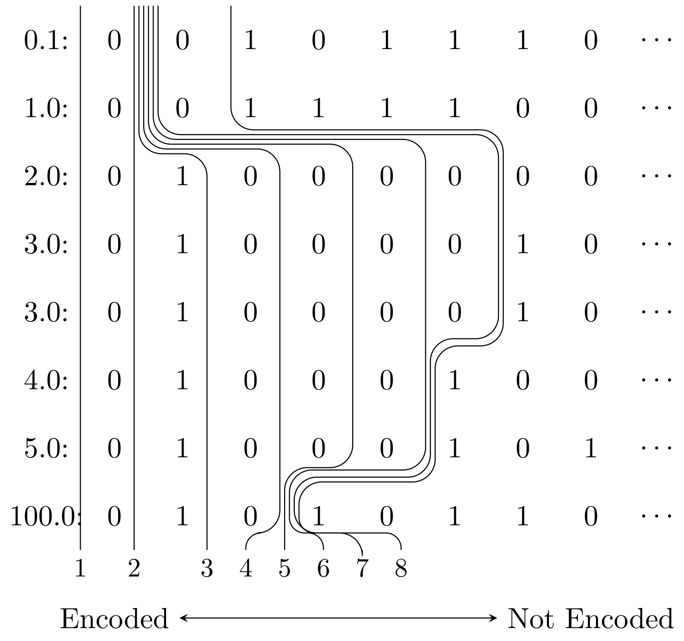
Many values share common prefixes. fzip uses arithmetic codes to encode these common prefixes.

First fzip creates a large tree to represent all the values in the array. The tree follows the following rules: each node has up to two children. Each edge represent a 1 bit or a 0 bit. Each node also has a weight which represents the number of values with the prefix the node represents. Each node in the tree represents a prefix. The root node represents “” or no prefix. So the weight of the root node equals the total number of values. The weight of the left (or “0” bit) child of the root represents the prefix “0”. Its weight represents the number of values that start with “0” (all non-negative values). Likewise the right child of the root represents the prefix “1” and its weight is the number of values starting with 1 (all the negative values).

Several properties appear. First, the sum of all the weights of the nodes in any level equals n, where n is the total number of values. Moreover the weight of any set of nodes that partitions the root node from the 65th level (and does not contain more nodes than necessary



(a) Each node in the above tree represents every prefix that occurs in the dataset.



(b) The above sorted list of values gives a second visual representation of how the partition grows.

Figure 6.4: The above 2 figures show the first 8 partition cuts for prefix compression for the example dataset {0.1, 1.0, 3.0, 5.0, 3.0, 100.0, 4.0, 2.0}. For simplicity half-precision (16-bit) encoding is used.

to create the partition) equals n.

Second, the tree is unbalanced (in our case this is good). Put another way, the datasets contain an unequal number of positive and negative numbers, also any "normal" dataset would not have an exponential distribution from 2^{-12} to 2^{12} in such a way to make the rest of the tree balanced.

Tree creation starts with a root node, which has a starting weight of 0. To create the rest of the tree, add each value to the tree in the following way: start by incrementing the weight of the root node. Let us also create a pointer to a "current node" c and initiate c to the root node. Then with the most significant bit (the sign bit) of the floating point value, update the current node by following the edge that matches this bit. If this edge does not exist create the edge and corresponding node. Then increment the weight of the new current node. This repeats until you reach the 64^{th} bit. Then the this process repeats for each value in the dataset.

fzip calculates the prefix codes by creating a partition in the tree. To start fzip creates a partition with only the root node. Then you include the edge with the largest cut length in the partition. This repeats until a predetermined number of edges become cut by the partition.

Each node added improves the compression because of the following observation: It takes $\log_2(k)$ bits to encode k edges in the partition. Take an edge (a_e) cut by the partition, which connects to child node a. When a gets added to the partition the number of bits in the not encoded stream will decrease by weight(a). However, each code will increases on average by $\frac{1}{k}$ (Each code has $\lceil \log_2 k \rceil$ bits and removing the ceiling and taking the derivative gets $\frac{1}{k}$). So the codes stream will increase by $\frac{n}{k}$. If you choose a to maximize weight(a) (a greedy algorithm) then weight$(a) >$ average edge cut $> \frac{n}{k}$.

You may notice a problem with this scheme. What if a value that occurs often, say 1.0 occurs %10 of the time? Ideally you should encode 1.0 as 4 bits ($\log_2 10$ rounded up), but if we continue to grow the partition beyond cutting 16 edges 1.0 would encode as more than 4 bits. Our solution freezes the codes once a node from the last (65^{th}) level becomes included in the partition. This allows fzip to continue to improve prefix compression by growing the partition and also encode common values with shorter codes. This change makes the encoding variable-length arithmetic encoding.

Of course the overhead to storing all of the codes exists. Currently a 16 byte record describes each code. Each record stores the prefix, the prefix length and the code length. To balance the benefit of prefix encoding with its overhead, we limit the overhead to %1 of the original size of the original array size (after BWT compression).

6.3.2 Repeated Value Extension

Prefix compression does not compress all of the repeated values. So fzip extends prefix compression to specifically include all repeated values. Again explaining why repeated values compress well: All of the datasets have less than 37 million values. An index of 26 bits can address the entire dataset. Even if a value repeats only once (occurs twice) there still exists an advantage to store the repeated values in a repeated value array and store the indexes into this array instead of the original values. In the previous example $26 + 26 + 64 < 64 + 64$ (2 indices plus the value in the array equals less than storing 2 values).

We add additional codes to the prefix codes to encode these repeats. To make room for the additional codes we increase the original code lengths by up to 1 bit each. This seems like a small trade off to make. To reduce overhead, the repeats only encode as 8 bytes (not 16 like the prefix codes), because fzip makes all the codes the same length for repeat compression.

We did develop Burrows-Wheeler transform compression for fzip, however we do not see this compression as hardware ameanable. Our current ideas for taking advantage of repeating patterns include a history table or a hash table.

6.4 Disscussion

We use fzip's results to get an idea of performance. We currently don't have the hardware ameanable version of fzip ready.

We also plan to switch the dataset from FPC's datasets to the matrix datasets.

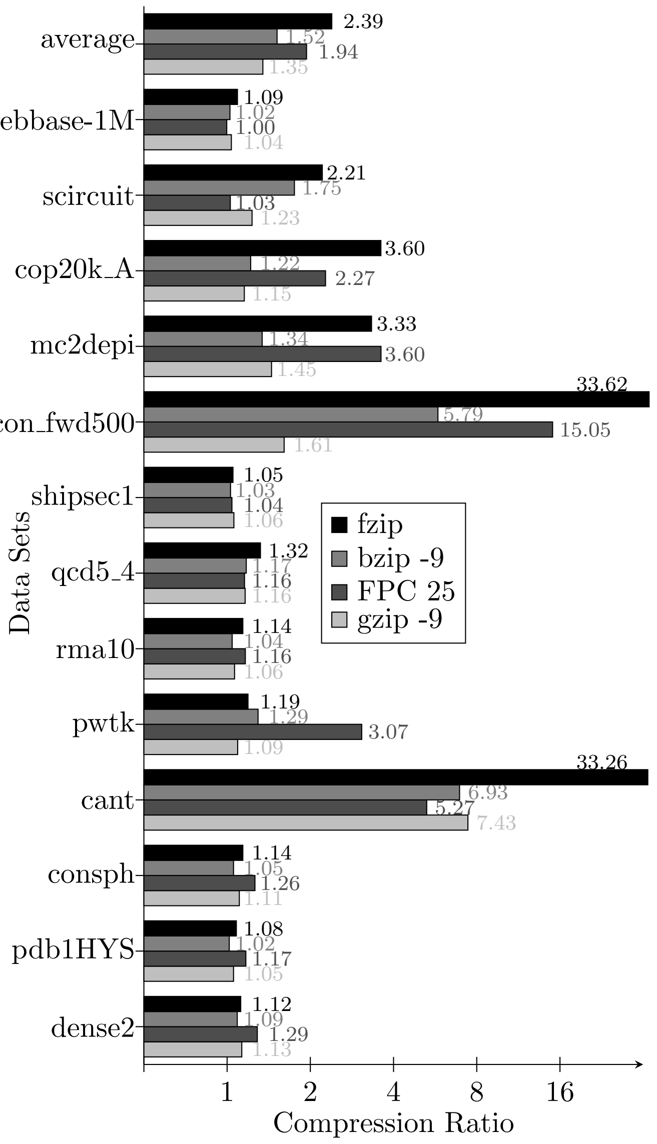


Figure 6.5: The comparison of different compression schemes shows fzip performs quite well.

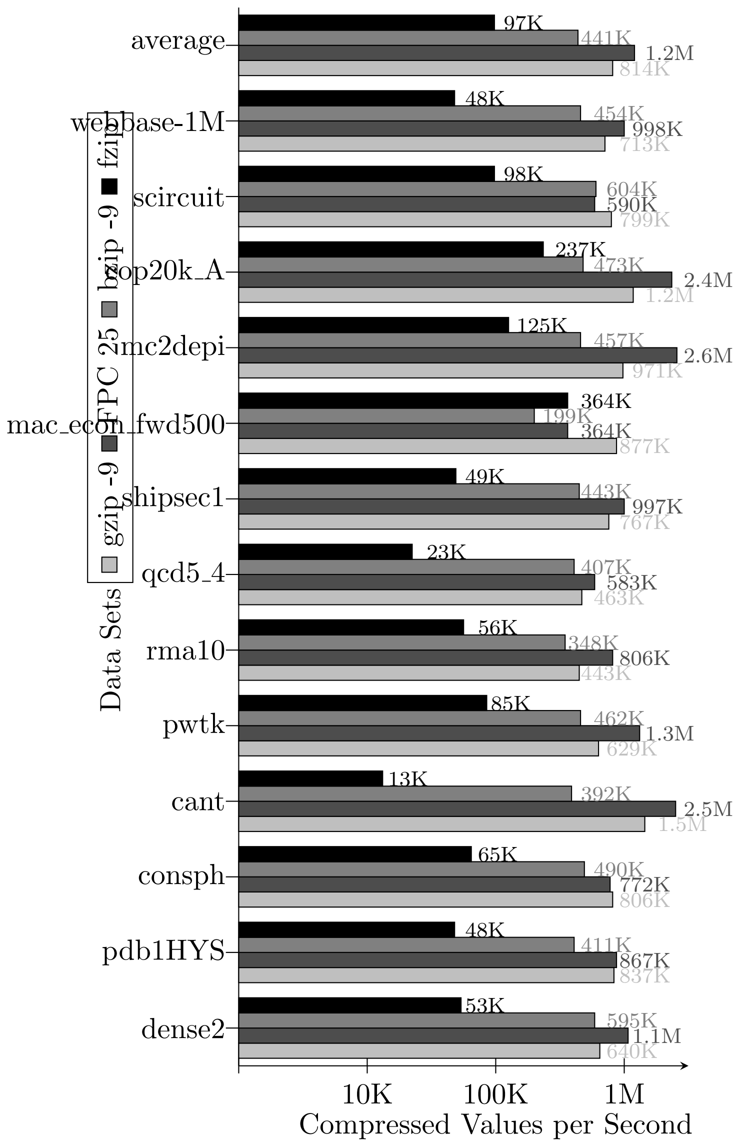


Figure 6.6: The above compression runtime analysis shows that fzip has some improvement to make to compete with other program's runtime.

Chapter7. MULTI-PORT RAM

Our multi-port RAM was created to allow mutliple processing elements to use a shared memory. It is a component in the larger design for a sparse matrix vector multiplier. Specifically it allows the decoder to access more memory space without going off chip. The component allows each PE to access a table that is 16 times larger than if it was stored inside each PE. Multi-port RAMs have been designed before, but none seemed to have the desired performance.

7.1 Design

The scratchpad is made up of several smaller components. The obviously being the need for block memory.

7.1.1 Memory Block

BlockRAMs are the efficient large memory blocks. 16 are used because of the larger space and the need to have 16 ports so that 16 memory operations can occur in parallel. We have multiple options to decide how the address space works. The simpliest would be to assign the first 16th addresses to memory0, the next 16th to memory1 etc. However, in our use case the lower addresses are used more, so this would lead to uneven access and therefore bottlenecks.

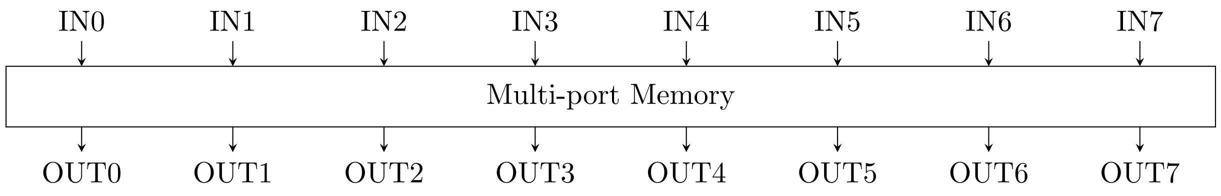


Figure 7.1: This model is impossible to achieve in real life. There is no way to scale the design by only $O(n)$. However there will be no problems simulating this design. It also gives us a clear goal of what we want to get as close to as possible.

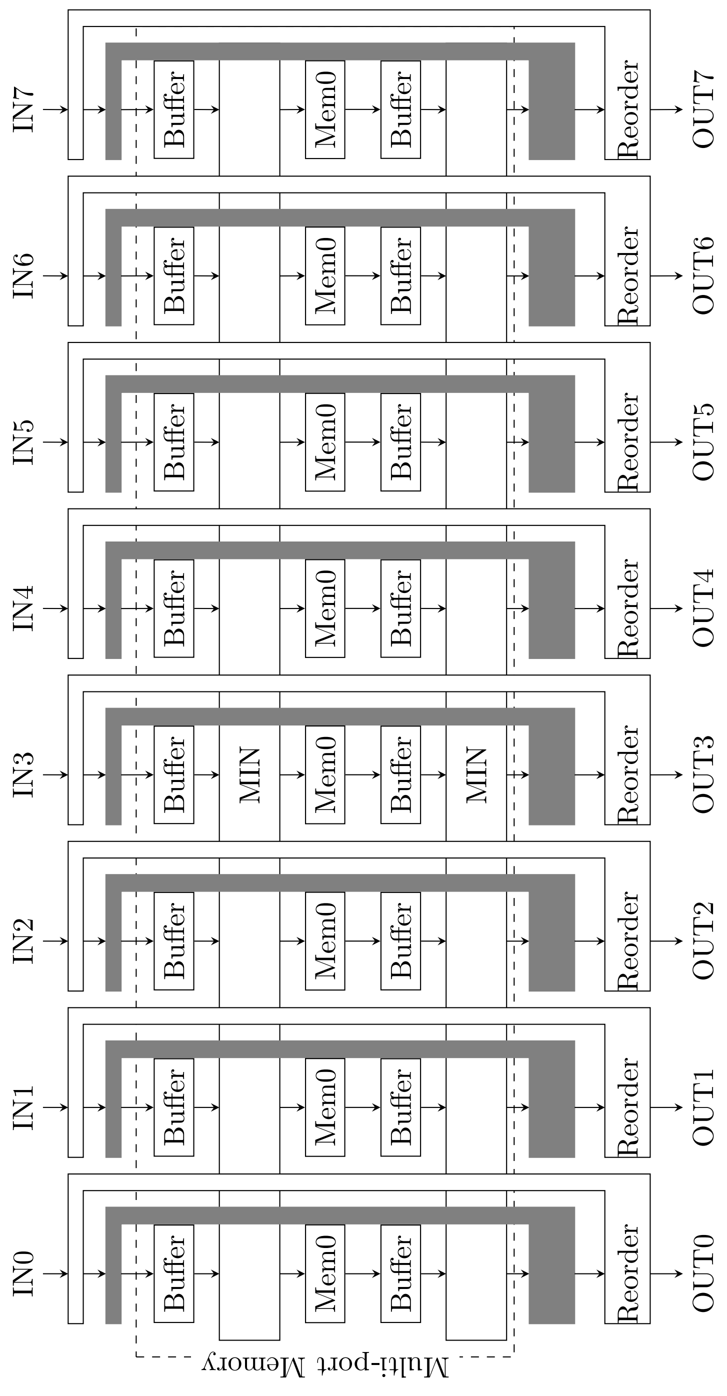


Figure 7.2: High-level view of f-scratch

So we use an interleaving memory address space.

7.1.2 Cross Bar

7.1.3 Reorder Queue

7.1.4 Tiny Cache

7.2 Conclusion

Chapter8. EXPECTED RESULTS

Chapter9. CONCLUSIONS

BIBLIOGRAPHY