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)

Program of Study Committee:

Joseph Zambreno, Major Professor

Phillip H. Jones

Chris Chu

Eric Cochran

Akhilesh Tyagi

Zhou Zhang

Iowa State University

Ames, Iowa

2015

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Hundreds of papers on accelerating sparse matrix vector multiplication exist, however, only a handful target FPGAs. Many have claimed that FPGAs inheriently perform infererorly to CPUs and GPGPUs. FPGAs do perform inferiorly for some applications like matrix-matrix multiplication and matrix-vector multiplication. CPUs and GPGPUs have too much memory bandwidth and too much floating point computation power for FPGAs to compete. However, SpMV trips up both CPUs and GPGPUs because of irregular memory access patterns, extra index information needed for each matrix element and irregular matrix patterns. We see this as a leveling of the playing field for FPGAs. Just as CPU and GPU implementations do, we create a matrix format specific to our implementation. This matrix format called R^3 format reorders matrix elements for better vector reuse and better compression. This format uses compression to effectively increase the memory bandwidth of the FPGA. This implementation uses a multiply accumulator capable of keeping several hundred intermediate values before outputting final result values.

CHAPTER 1. 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. Eigenvector estimation often uses iterative SpMV operations. For example, the pagerank algorithm uses SpMV for eigenvector estimation.

For the most part modern CPUs perform SpMV well. Some may even say FPGAs deserve 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 repetitive 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, CPU 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 to 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.

CHAPTER 2. 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 1 million people, an impressive feat. In other words, from the time you started reading this paper 100 people have joined facebook and are not friends with you. The average user has 300 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 roughly 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. By convention indices are 4 bytes (32-bit) integers. Values are either single-precision (32-bit) or double-precision (64-bit) floating point values. 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, 3, 4, 4, 4, 4, 5, 5, 6, 6, 6, 6, 7, 7, 7, 7

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

VALUE: A_{11}, A_{14}, A_{17}, A_{25}, A_{28}, A_{32}, A_{33}, A_{36}, A_{37}, A_{41}, A_{45}, A_{53}, A_{54}, A_{57}, A_{58}, A_{62}, A_{65}, A_{72}, A_{73}, A_{76}, A_{78}, A_{83}, A_{84}, A_{85}, 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 simple and convenient, but not a necessary way to traverse 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 (number of non-zero elements per row) 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 200 GFLOPS for matrix matrix multiplication and 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 200 GFLOPs, 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.

Several optimizations to improve the performance of SpMV exist. We cover CPU and GPU optimizations first. As we cover techniques we also look at how they could apply to FPGA implementations. If you are impatient feel free to skip ahead to the section about SpMV on the FPGA.

2.3 Compressed Sparse Row Format (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. CSR format stores the first element of each row instead of the row index of each element. 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. 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 relies on the previously 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 does not 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 in equation 2.1 is shown:

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

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

VALUE: A_{11}, A_{14}, A_{17}, A_{25}, A_{28}, A_{32}, A_{33}, A_{36}, A_{37}, A_{41}, A_{45}, A_{53}, A_{54}, A_{57}, A_{58}, A_{62}, A_{65}, A_{72}, A_{73}, A_{76}, A_{78}, A_{83}, A_{84}, A_{85}, A_{86}

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 sub-blocks.

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 is usually coupled with a second matrix; meaning the matrix is the sum of 2 matrices one in BSR format the other in CSR or COO. Formats that use the sum of two smaller matrices are called a hybrid format. We have pessimistic view of hybrid formats, because this results in performing SpMV on 2 matrices sparser than the original. The rest of the field agrees with this and tries to minimize this negative effect by minimizing the size of the second matrix.

The block sparse for the example in equation 2.1 is shown:

ROW: 0, 0, 2, 2, 4, 4, 4, 6, 6, 6

COLUMN: 3, 6, 0, 4, 1, 3, 6, 1, 3, 5

Value: $\{A_{14}, 0, 0, A_{25}\}$, $\{A_{17}, 0, 0, A_{28}\}$, $\{0, A_{32}, A_{41}, 0\}$, $\{0, A_{36}, A_{45}, 0\}$, $\{0, A_{53}, A_{62}, 0\}$, $\{A_{54}, 0, 0, A_{65}\}$, $\{A_{57}, A_{58}, 0, 0\}$, $\{A_{72}, A_{73}, 0, A_{83}\}$, $\{0, 0, A_{84}, A_{85}\}$, $\{A_{76}, 0, A_{86}, 0\}$

Secondary COO Matrix:

ROW: 0, 2, 2, 6

COLUMN: 0, 2, 6, 7

VALUE: A_{11}, A_{33}, A_{37}, A_{78}

This example does not actually save space because of the extra 0s stored, however, bitmaps

can be used instead storing explicit zero values.

2.5 Cache Blocking

CPU optimizations also include changing the matrix traversal for better vector reuse. BSR does this to a small extent. One method called Cache blocking traverses large sub-blocks 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 (Chapter 5). The Cache Blocking in COO format for the example in equation 2.1 is shown:

ROW: 0, 0, 2, 2, 3, 0, 1, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 7, 4, 4, 5, 6, 6, 7, 7

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

VALUE: A_{11}, A_{14}, A_{32}, A_{33}, A_{41}, A_{17}, A_{25}, A_{28}, A_{36}, A_{37}, A_{45}, A_{53}, A_{54}, A_{62}, A_{72}, A_{73}, A_{83}, A_{84}, A_{57}, A_{58}, A_{65}, A_{76}, A_{78}, A_{85}, A_{86}

2.6 GPU Processors

Before discussing storage formats specific to GPUs, it is important to understand GPUs play the computation game differently the CPUs. To show this let us compare a high end CPU (Intel Xeon E7-8890) and a high-end GPU (Nvidia Tesla K40). The GPU has a max throughput of 1.66 TFLOPS (double precision). The CPU has a max throughput of 408 GFLOPS (double precision). The GPU has 1.5MB of cache. The CPU has 38MB of cache. The cache is growing every generation as well. The previous Tesla (Fermi) had 768KB of cache. The previous Xeon had 15MB of cache. The GPU is a vector processor making it hard to get good performance on unstructured computation. The K40 supports up to 26000 threads whereas the E7-8890 supports 30 threads.

When using COO format each thread processes n values. Some synchronization occurs to ensure the correct y values are stored. This kernel performs relatively well due to the good load balancing. However, this format does hardly any x vector reuse. In fact, if you disable the cache, you get almost identical performance.

In CSR format the GPU assigns a thread or group of threads per row. This method achieves

much better vector reuse and therefore better performance. One way to think about this is that all the threads start by processing values on the left side of the matrix and proceed to the right. This means different threads will process elements with the same column index at around the same time, leading to x values being reused before getting flushed from the cache.

2.7 ELLPACK

To enable better performance **?**) used a storage format 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.

It is a little tricky to explain why ELLPACK works so well. First, each processor works on one y value at a time. Second, loop unrolling is very easy. Third, the extra processing on zero values is not that much of a waste considering that this application is memory bound. Fourth, and this is our reason, the Tesla K40 supports running 26000 threads. This means the GPU will have roughly 2600 intermediate y values. The computation for each thread starts at the beginning of each row. This means that many of the threads will request the same x values at round the same time. This results in fewer cache misses.

The ELLPACK formate for the example would be:

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

VALUE: A_{11}, A_{14}, A_{17}, 0, A_{25}, A_{28}, 0, 0, A_{32}, A_{33}, A_{36}, A_{37}, A_{41}, A_{45}, 0, 0, A_{53}, A_{54}, A_{57}, A_{58}, A_{62}, A_{65}, 0, 0, A_{72}, A_{73}, A_{76}, A_{78}, A_{83}, A_{84}, A_{85}, A_{86}

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

2.8 Block-ELLPACK

The Block-ELLPACK format for the example in equation 2.1 is shown below:

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

VALUE: $\{\{A_{14}, 0, 0, A_{25}\}, \{A_{17}, 0, 0, A_{28}\}\}$, $\{\{0, A_{32}, A_{41}, 0\}, \{0, A_{36}, A_{45}, 0\}\}$, $\{\{0, A_{53}, A_{62}, 0\},$

$\{A_{54}, 0, 0, A_{65}\}\}$, $\{\{A_{72}, A_{73}, 0, A_{83}\}, \{0, A_{76}, A_{85}, A_{86}\}\}$

Secondary COO Matrix:

ROW: 0, 2, 2, 4, 4, 6, 7

COLUMN: 0, 2, 6, 6, 7, 7, 3

VALUE: A_{11}, A_{33}, A_{37}, A_{57}, A_{58}, A_{78}, A_{84}

2.9 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 work in **?**) does a reasonable and created a 144 GFLOPS engine on a Virtex-7 X690T. However, they over utilize DSP blocks by 40% by using them for addition without using the 25×18 multiplier.

The basic idea of an FPGA is to design the processor you want and that design can be loaded on to an FPGA. Since CPUs and GPUs suffer from bad SpMV performance it seems possible to design a SpMV processor to load on an FPGA and get better performance.

For example RAM blocks are distributed equally across the chip meaning that block RAMs can be located in a multiply-accumulator storing intermediate y values. In a CPU the cache is a far distance from the ALU and it does not make as much sense to store many intermediate y values.

2.10 Column Row Traversal

We view storing intermediate y values as a superior way to reduce the number of vector requests, over caching vector values. For example let us compare 2 strategies.

First, row traversal and the last 4 vector values are stored in cache. In this scheme cached values only get used twice the first is a value A_{72}.

Second, column traversal for every 4 rows. In other words 4 intermediate y values are stored. This traversal in COO format would be:

ROW: 0, 3, 2, 2, 0, 1, 3, 2, 0, 2, 1, 5, 6, 4, 6, 7, 4, 7, 5, 7, 6, 7, 4, 4, 6

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

VALUE: A_{11}, A_{41}, A_{32}, A_{33}, A_{14}, A_{25}, A_{45}, A_{36}, A_{17}, A_{37}, A_{28}, A_{62}, A_{72}, A_{53}, A_{73}, A_{83}, A_{54}, A_{84}, A_{65}, A_{85}, A_{76}, A_{86}, A_{57}, A_{58}, A_{78}

2.11 Delta Compression

Another index compression scheme, delta compression, compresses indices more aggressively than BSR by storing delta distances. A delta is the distance between the previous and current matrix element. The average number of bits to store a delta value is quite small (discussed in chapter 5). **?**) is vage on the details, however since this uses variable length encoding (VLC) some encoding scheme is required. This saves a lot of space, but the results are mediocre. 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. The delta codes for the example in equation 2.1 is shown:

DELTAS: 0, 2, 2, \n, 4, 2, \n, 1, 0, 2, 0 \n, 0, 3, \n, 2, 0, 2, 0, \n, 1, 2, \n, 1, 0, 2, 1, \n, 2, 0, 0, 0

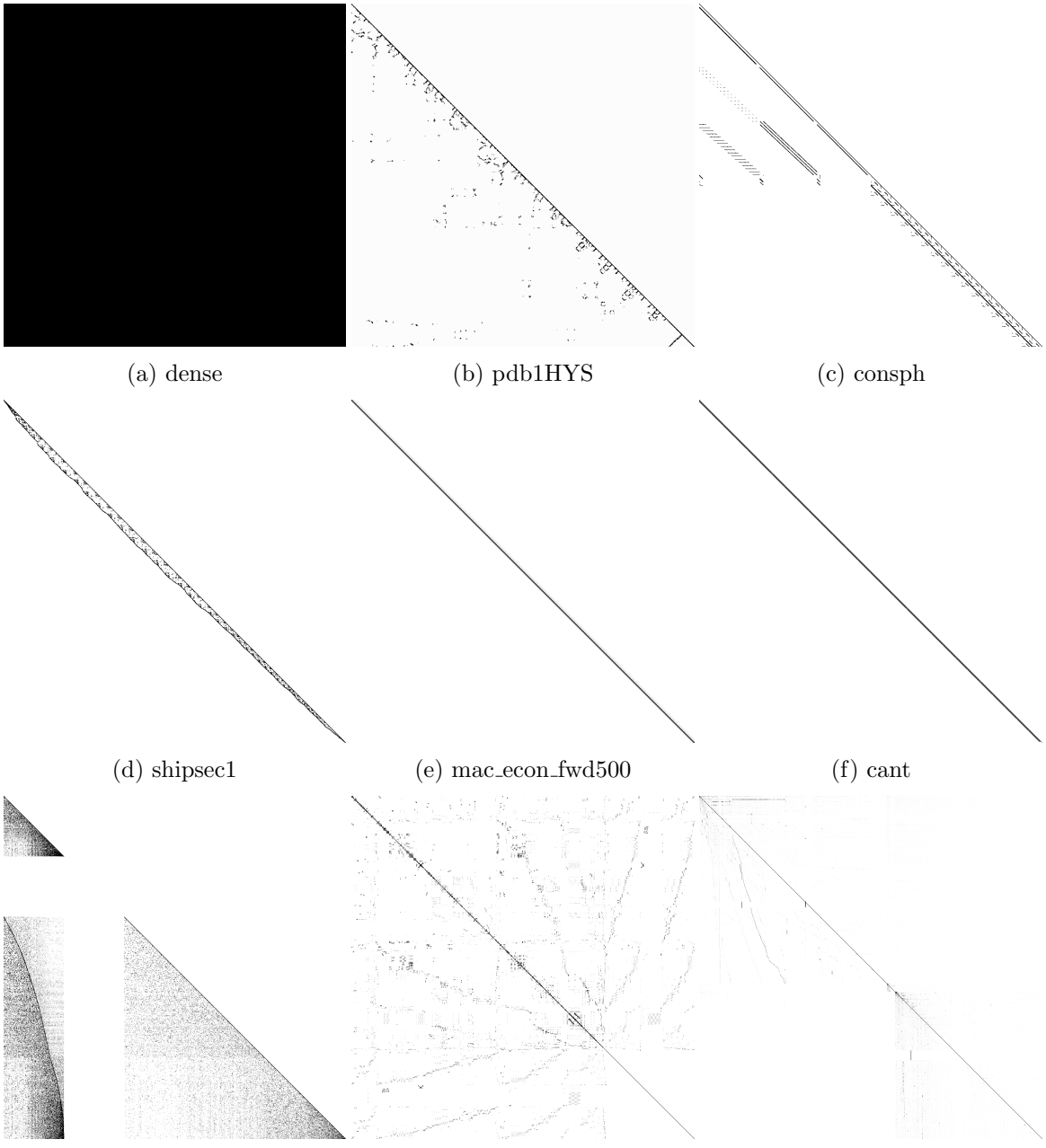
TODO: talk about encoding

2.12 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 mediocre. Again, FPGAs can dedicate area for decoder.

2.13 Benchmarking

Ok, now we have a good background about SpMV, the platform it can run on and optimizations for SpMV we need a way to determine who is the best. This when benchmarking comes in. However, different matrices can have vastly different performance. In figure 2.4 we



(a) dense (b) pdb1HYS (c) consph

(d) shipsec1 (e) mac_econ_fwd500 (f) cant

(g) cop20k_A (h) scircuit (i) webbase-1M

Figure 2.1: The density plots of the matrices used for testing

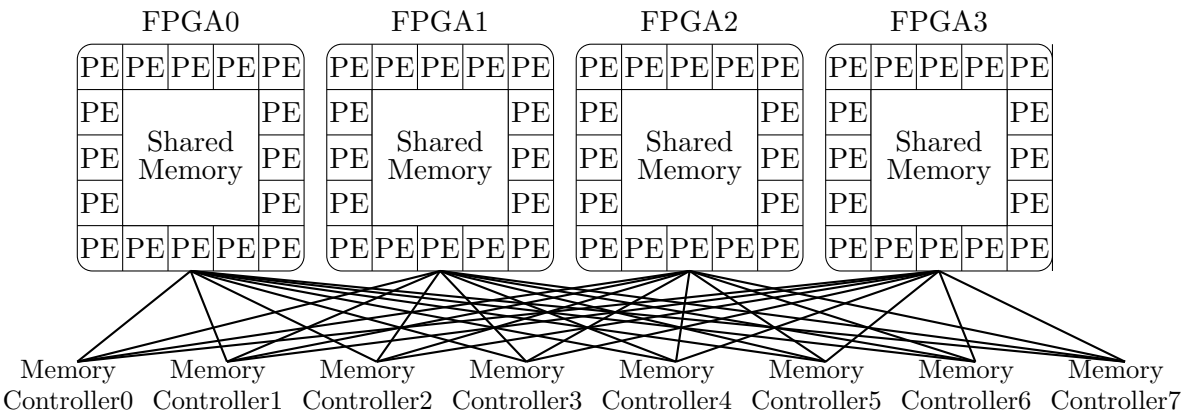


Figure 2.2: R^3 implementation on the Convey HC-2 coprocessor: 4 Virtex-5 LX330 FPGAs tiled with 16 R^3 SpMV processing elements (PE) each. Each Virtex-5 chip connects to all 8 memory controllers, which enables each chip to have access to all of the coprocessor's memory.

show the performance of SpMV on CPUs, GPUs and FPGAs. As you can see the performance is very jumpy from matrix to matrix. 3 factors effect the performance: dimension, sparsity, and values.

The dimension of a matrix are the height (M), the width (N) and the number of nonzeros (nnz). These metrics effect different processors differently.

For CPUs the nnz and N are important values. As figure 2.4 shows when the matrix no longer fits in cache it takes a performance hit. It takes a second performance hit (not shown) when the width of the matrix and the therefor the length of the x vector grows to the point when the x vector also can not fit in cache.

For GPUs cache plays less of a role. However, two factors conspire against GPUs. The matrix formats they use and the amount of RAM on GPU boards. The best performing matrix formats of GPUs, ELLPACK and Block-ELLPACK, also introduce 0 values and take up the most memory space. GPU boards currently have at most 12GB of on board RAM compared to the 128 or more possible on CPUs. This means as matrices approach and go beyond 1 billion values then GPUs have to use worse performing matrix formats or be completely unable to perform SpMV.

TODO: FPGAs TODO: sparsity TODO: values

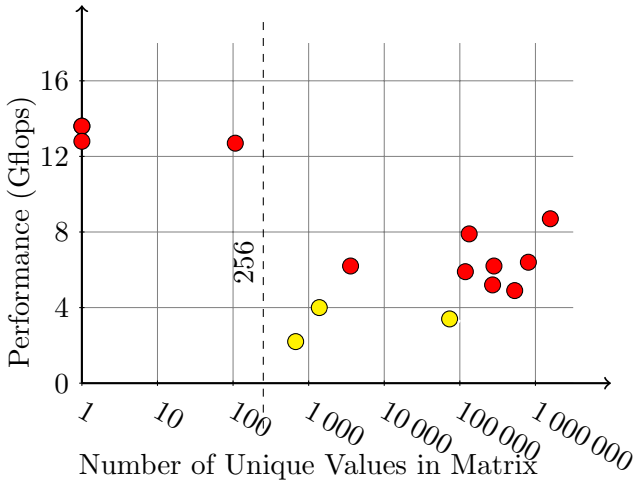


Figure 2.3: Unique values in a matrix vs the performance of R^3. Matrices with fewer than 256 unique values (only common elements exist) enables R^3 format to compress much better. The ●'s are outliers due to their size (see Figure 2.4).

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

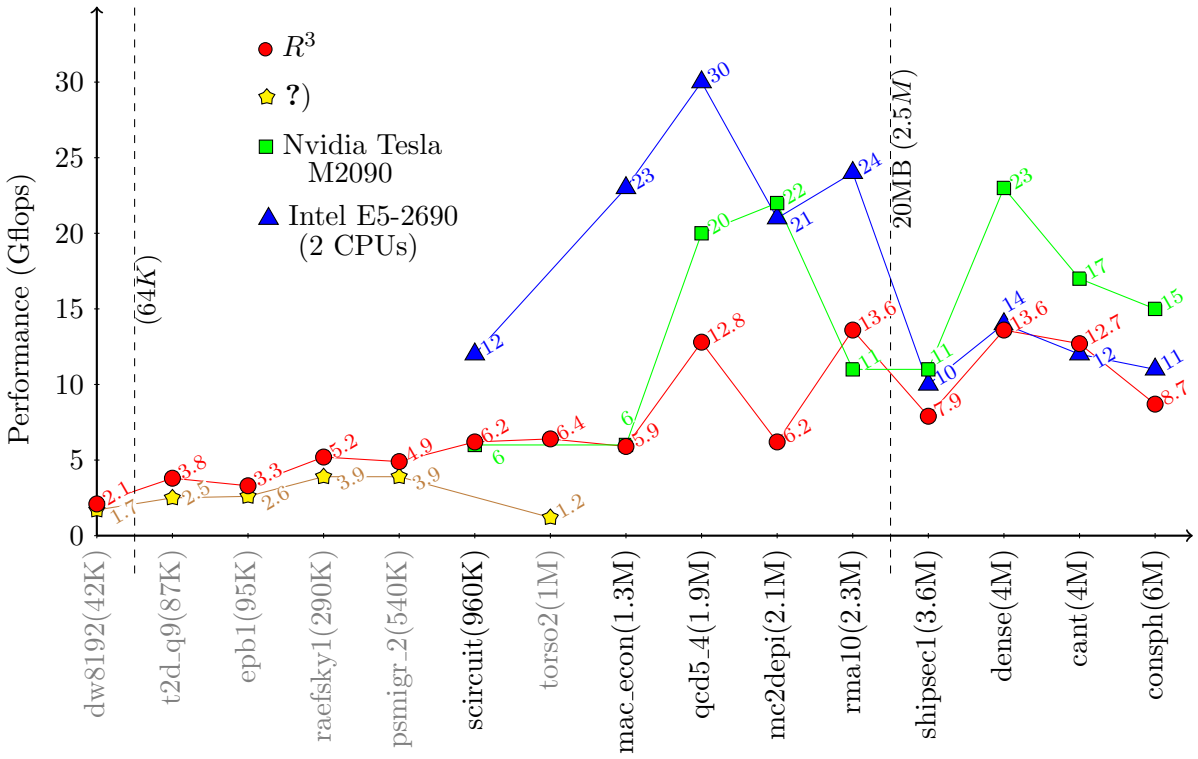


Figure 2.4: *nnz* vs Performance on each platform. The small matrices, ones around 64K or less, performed poorly on R^3, due to the overhead. CPUs experience the opposite effect. They take a performance hit once the matrix no longer fits in cache.

CHAPTER 3. SpMV on FPGA METHODOLOGY

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

Our design methodology consists of 3 design pillars. The first pillar is the design of a multiply accumulator that does not stall and maintains multiple intermediate values. The second pillar is the design of a matrix traversal that enables reuse of vector values and improves matrix compression. The third pillar is the design of a matrix compression scheme with a high compression ratio and has a hardware amenable 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 also should not stall, or at least 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 preceding matrix element in the traversal.

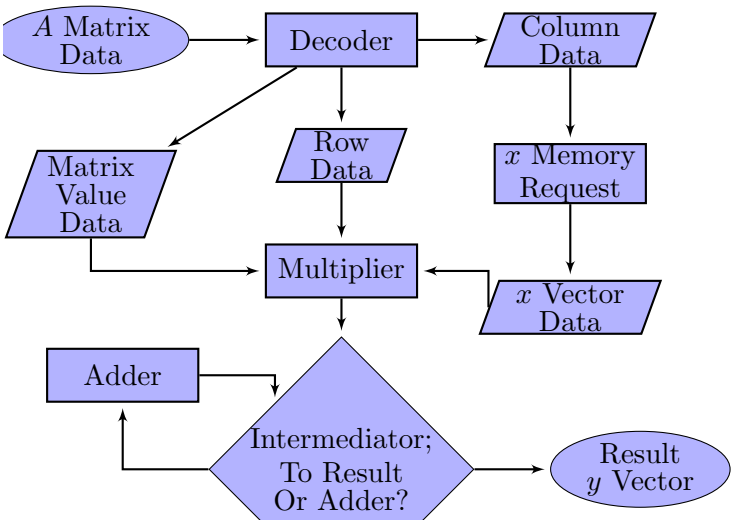


Figure 3.1: Data flow of an R^3 SpMV processing element. The processing element needs the column data before accessing the vector data.

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 SpMV is usually used for iterative and repetitive 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 boundaries. We give little regard to boundaries 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 bandwidth. Values repeat more than one would expect in matrices. Taking advantage of this repetition is the biggest step towards good compression. Figure 3.3 shows how 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 3.1). The architecture diagram (Figure 3.2) also follows the same general flow of the dataflow diagram.

We separate our current and planned contributions into 5 pieces, the next 5 chapters. Chapter 3 focuses entirely on the first pillar, the multiply accumulator design. Chapter 4

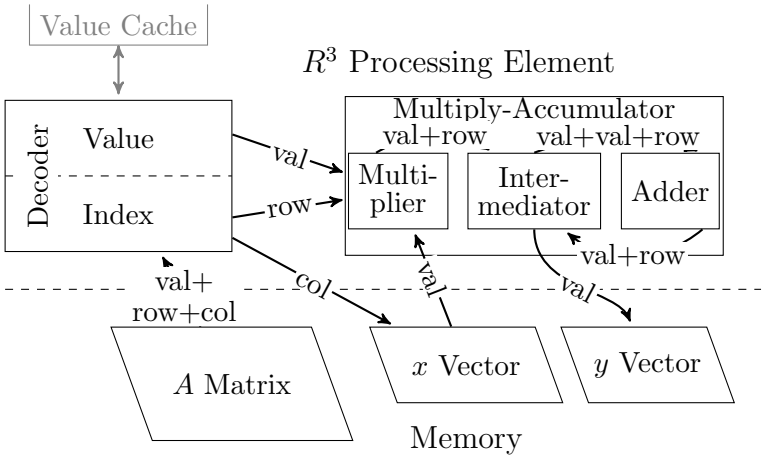


Figure 3.2: A single R^3 processing element. The arrows show the flow of data through the processing element. Although this diagram shows the memory access to each of the 3 places in memory as separate, they share one memory port. The diagram also does not show the FIFOs that help keep the pipeline full.

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

CHAPTER 4. MULTIPLY-ACCUMULATOR

A high throughput SpMV implementation relies on designing a no-stall multiply accumulator (MAC). An inefficient engine stalls when a matrix and its associated vector value 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 [**?**)].

In R^3 [**?**)] we designed a block called an Intermediator (Section 4.1) capable of storing 32 intermediate y vector values. In our next design we intend 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 the traversal. The rest of the chapter discusses the new design. The matrix elements in one set

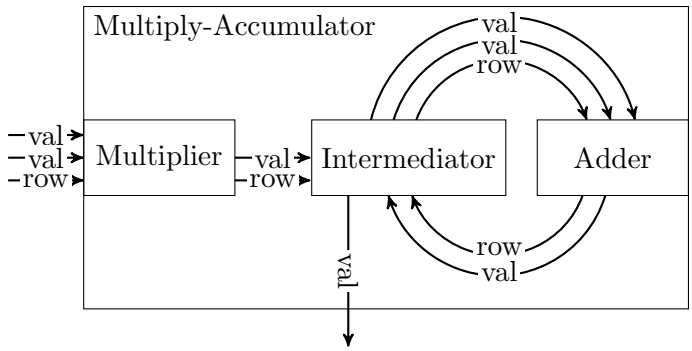


Figure 4.1: The no-stall multiply-accumulator block handles multiple intermediate values at a time. This allows multiple intermediate values in the adder pipeline.

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 that follows this rule called row column row (RCR) traversal.

4.1 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 multi-pump the distributed RAM to achieve two writes every clock cycle. The second option is to create a dual port RAM with Banking. Banking is discussed in chapter 9. The third option is to use 4 RAM blocks and clever use of XORs.

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 being 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 512 clock cycles (1 element per row in the matrix for those 512 rows corresponding to the active window). It also takes 512 cycles to store the green "ready to store" elements. 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.

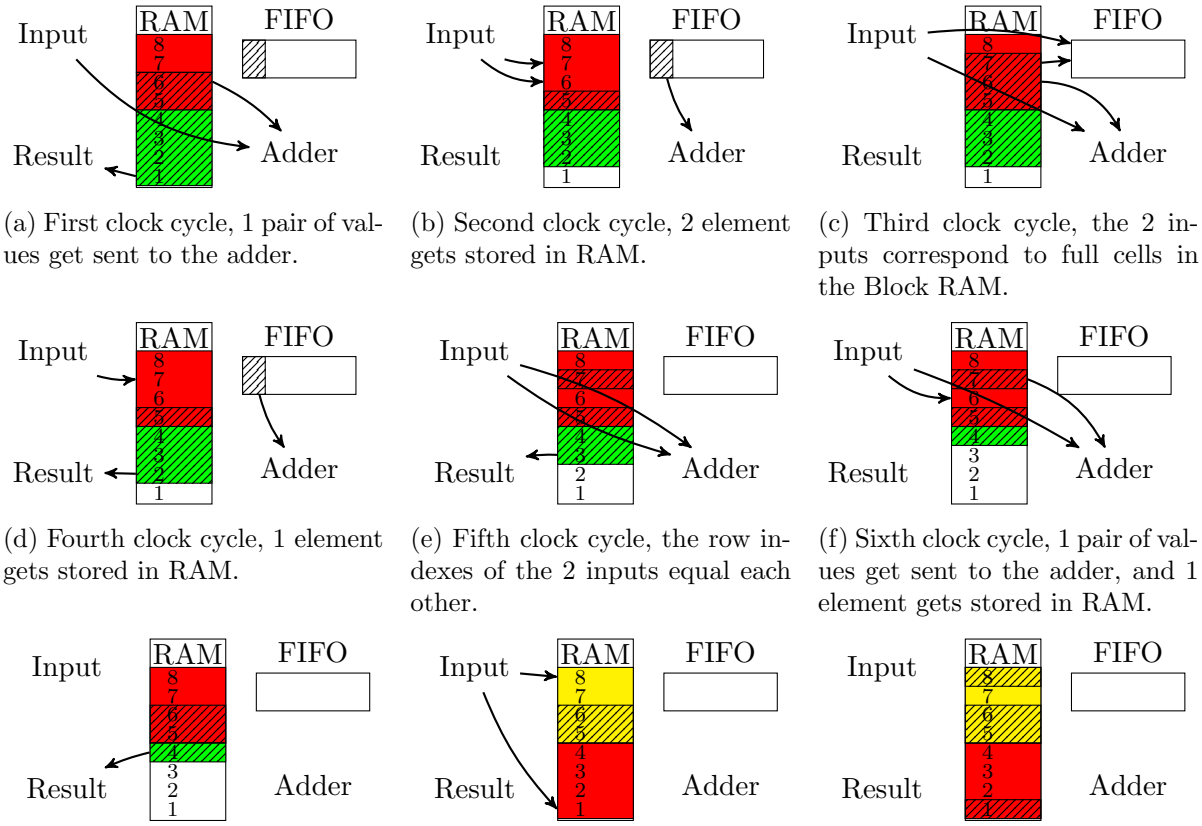
Case 2: (Figure 4.2d) Only one value arrives (valid) and the row corresponds to an empty cell. The value goes into the empty cell. If the "to result" window has values, it outputs a result, and if the FIFO has values it outputs a set to the adder.

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, 4.2i) 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.

(i) 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 9 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. We assume each row has at least 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 the 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 (16%). This worst case behavior does not occur in practice.

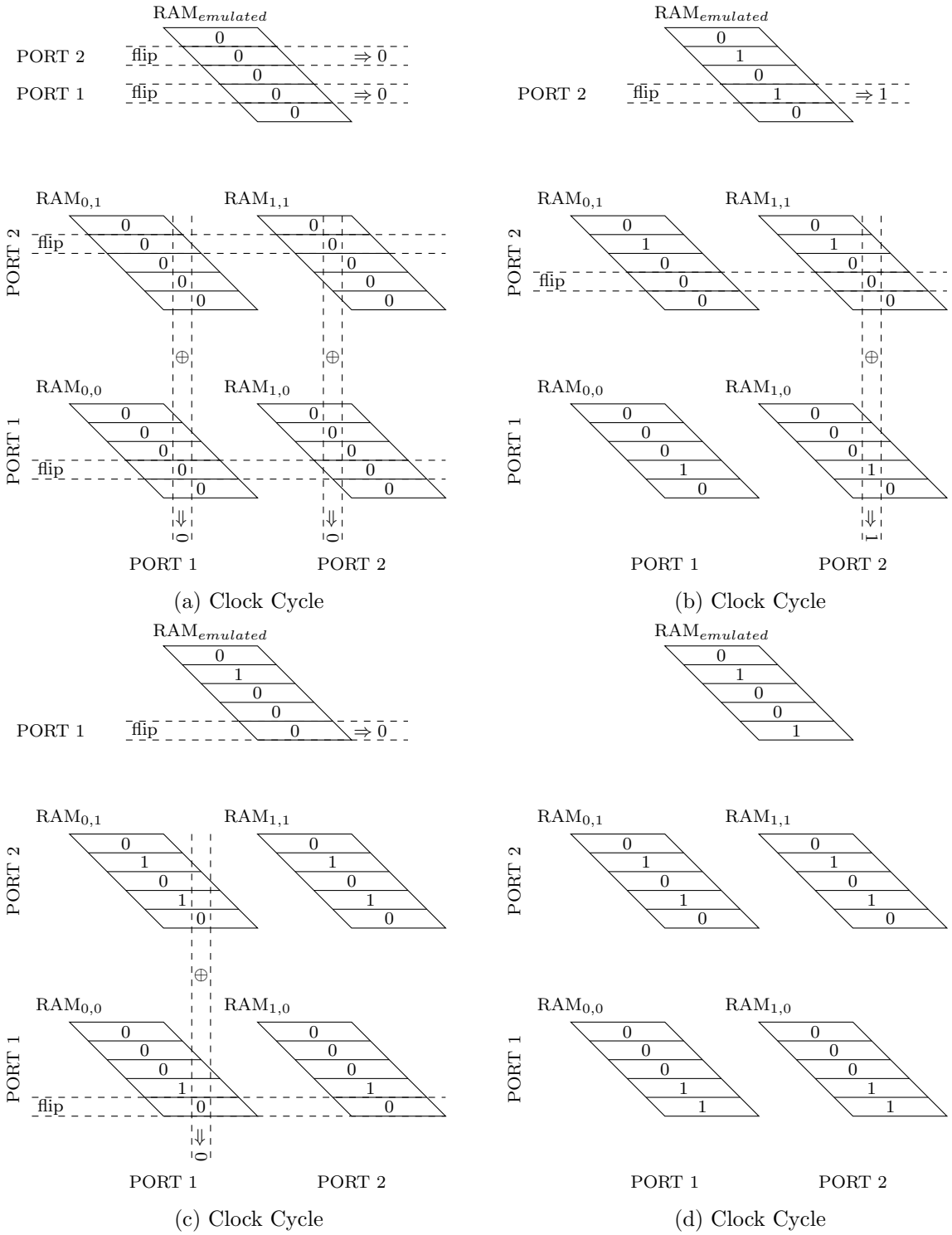
4.2 A Dual Port 1×1024 RAM with Zero Clock Cycle Latency

This trick is essentially a special case of). This requires the use of pseudo dual port distributed RAMs. Before looking at the implementation let us look at the target behavior. During an intermidiator status request the bit of the requested address will always flip. (Empty cells become full and full cells become empty.) This flip occurs after the correct status is reported.

FPGA vendors do not provide dual port distributed RAMs. Instead, they provide pseudo dual port distributed RAMs. That is RAMs with one read port and one write port.

With a clever arragement of 4 pseudo dual port RAMs we can emulate one dual port RAM. To begin with, arrange the RAMs in a 2×2 grid. The write ports of the 2 RAMs in each row are connected together. The read ports of the 2 RAMs in each column are connected by an XOR gate. The address on port 1 controls the address of the write port of the bottom row of RAMs. The address on port 1 also controls the address of the read ports of the left column of RAMs. Similarly, the address on port 2 controls the address of the write ports of the top row of RAMs. The address on port 2 also controls the address of the read ports of the right column of RAMs.

This may make more sense with the example in Figure 4.3.



(a) Clock Cycle

(b) Clock Cycle

(c) Clock Cycle

(d) Clock Cycle

Figure 4.3: The operation of the clever architecture.

CHAPTER 5. MATRIX COMPRESSION

Not much work focuses solely on matrix compression. However, matrix compression for SpMV has 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 (GRMLCM). For the rest of the paper we call this traversal column row traversal.

As mentioned in the background section, 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 use the name column row traversal. 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: $[A_{11}, A_{41}, A_{32}, A_{33}, A_{14} \ldots]$.

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.

5.2 Delta Compression

?) also noted the clumpy distribution of values in most matrices. So instead of storing the

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)

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 paper R^3 we used column-row-16. In the previous matrix example the deltas are: $[0, 3, 15, 3, 1 \ldots]$. R^3 stored the delta in a space between 5 and 45 bits (5, 13, 21, 29, or 45 bits).

Many papers use delta compression **???**). 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 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

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

occurs is because large delta values can represent a short vertical jump. (We start to see the disadvantage of row major traversal.) 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.

5.3 Row Column Row (RCR) traversal

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.

5.4 Encoding deltas

The easiest trend to see from the distribution table 5.2 is that more than half of the elements usually occur immediately after another element. So we choose to simply encode these as a 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

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

6 bit deltas and an 6 bit delta can both be encoded as an 7 bit delta. This wastes 1 bit of the 6 bit delta encoded as 7 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 table 5.2 shows the performance of this scheme.

CHAPTER 6. FLOATING POINT COMPRESSION

Floating point compression is the second half of matrix compression. Figure 6.1 shows a comparison of compression schemes. In the end, we created a program (and library) called fzip. In total fzip takes advantage of three compressible features of datasets: repeating sequences of values (patterns larger than 8 bytes long), repeating values (patterns exactly 8 bytes long), and repeating prefixes (patterns less than 8 bytes long).

For SpMV we need to make a hardware decoder. Currently everything except the BWT should be hardware amenable. We can remove that part and still expect good compression. Removing BWT removes compressing patterns larger than 8 bytes long. In the future we could use methods like LZW), which is a little bit easier to get into hardware.

6.1 Previous work

It was noted in **?**) 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 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 **?**) describing it's own compression scheme FPC. FPC performs well. 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 being able to look up previously occurring 8-byte values.

Table 6.1: Detailed value compression analysis and performance comparison

Matrix	uncompressed	Unique Values	Unique/nnz $\times 8$	256 Common	GZIP	FPC	Pattern Compression Unlimited	Pattern Compression 32KB
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

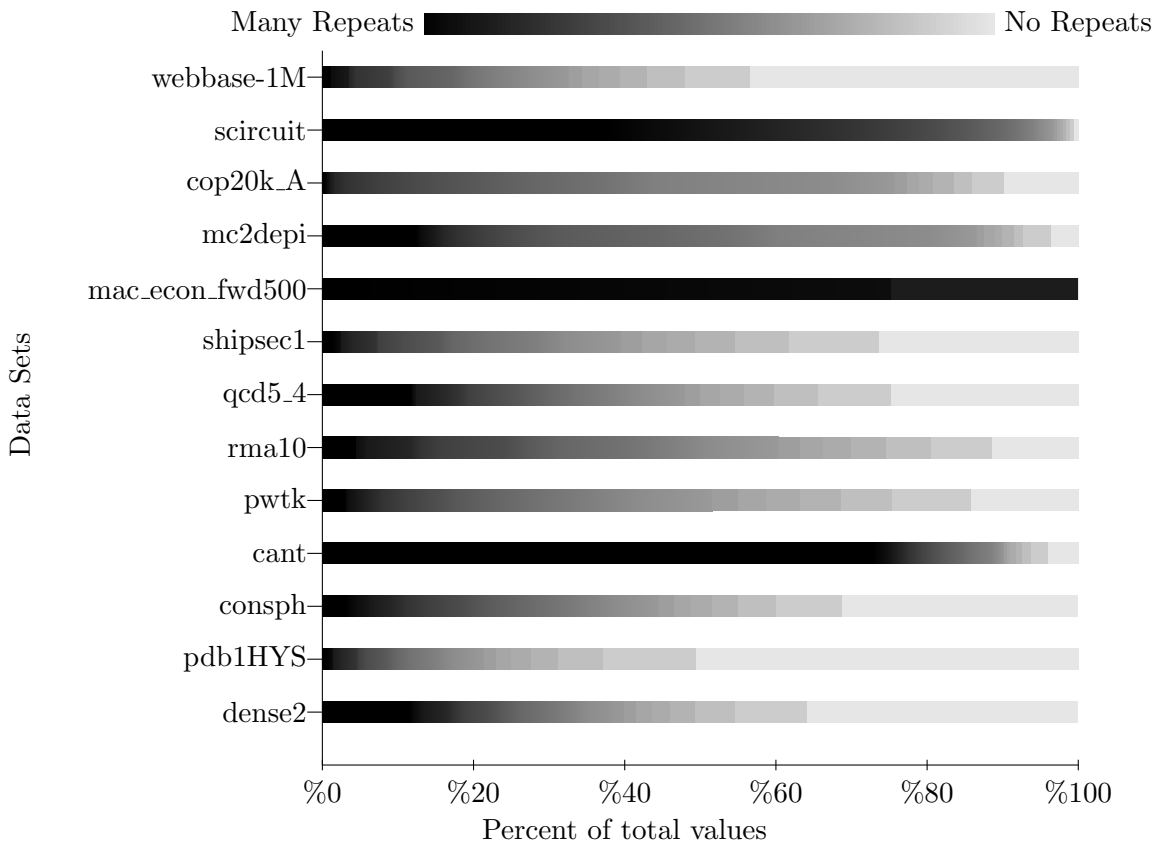


Figure 6.1: The above figure shows the distribution of repeats in each dataset. Each shade represents a different number of repeats. For instance: ■:> 512, ■:16, ■:2, ■:1(no repeats).

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 7.5), our approach to floating point compression (section 6.4) and our results (section **??**).

6.2 Related Work

FPC, the program most similar to our work, uses predictors to compress data. A predictor uses the previous data in the data set to guess the current element in the data set. FPC uses hash functions to implement their predictors. Passing an argument between 1 and 25 to FPC

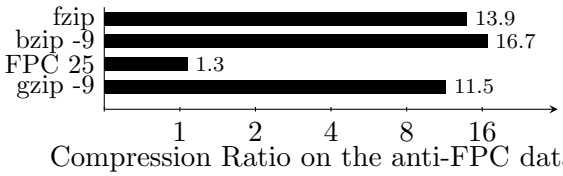


Figure 6.2: We engineering a dataset to make the performance of FPC look bad compared to other programs. Although unfair, this shows a type of pattern that FPC does not exploit and other programs do. This problem exists because FPC only uses predictors for compression.

configures the hash table size. Larger hash tables necessarily predict better because values in the hash table get overwritten less often. However, large hash tables cause slower performance compared to hash tables that can fit in cache.

In FPC, each value gets encoded with a 4 bit header followed by 0, 1, 2, 3, 4, 6, 7 or 8 bytes. The first bit specifies which of the 2 predictors resulted in the most matching bytes compared to the correct current value. Then, the next 3 bits encode this number of bytes, however, 5 bytes is rounded down to 4 bytes. Then, the least significant bytes that the hash failed to predict follow.

FPC has the advantage of speed. Particularly when the hash table fits in cache. Although good, the compression ratio suffers because predictors do not always get the best or a good prediction. If a pattern does not exist among the values in the data set then FPC can not predict with any accuracy. Let us design this "anti-FPC" dataset. Take the set of numbers $\{10^0, 10^1, 10^2, 10^3, \ldots, 10^9\}$ and randomly choose one (with replacement) to add to the anti-FPC dataset. We do this a million times to get a dataset with a million values (8MB in size). You can observe the performance of this in Figure 6.2.

6.3 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 will 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 in the dataset often share the same sign. (Some datasets only contain positive numbers.) Likewise, close values 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}.

We gauge the strength of the pattern in a particular dataset by looking at how many prefix bits the adjacent values share. Figure 6.3 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**?**) to analyze these patterns. Figure **??** describes this algorithm some, however, many other sources describe this algorithm in more detail**??**). Figure 6.4 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.

6.4 Our Approach

Our approach takes advantage of the features in Section 7.5. The algorithm starts with BWT compression, then compresses further with using prefix and value compression.

6.4.1 Burrows-Wheeler Transform Compression

After completing the Burrows-Wheeler Transform, fzip uses a simple encoding scheme (Figure **??**). For each value in the BWT, fzip pushes a '0' or '1' onto a bit array to denote whether the value equals the previous value. If the values differ (a '0' in the bit array) a second array stores the next value. The 4 datasets (msg_sppm, num_plasma, obs_error, obs_info) expected to do well from the analysis in Figure **??** do perform well under this compression scheme (see Figure **??**).

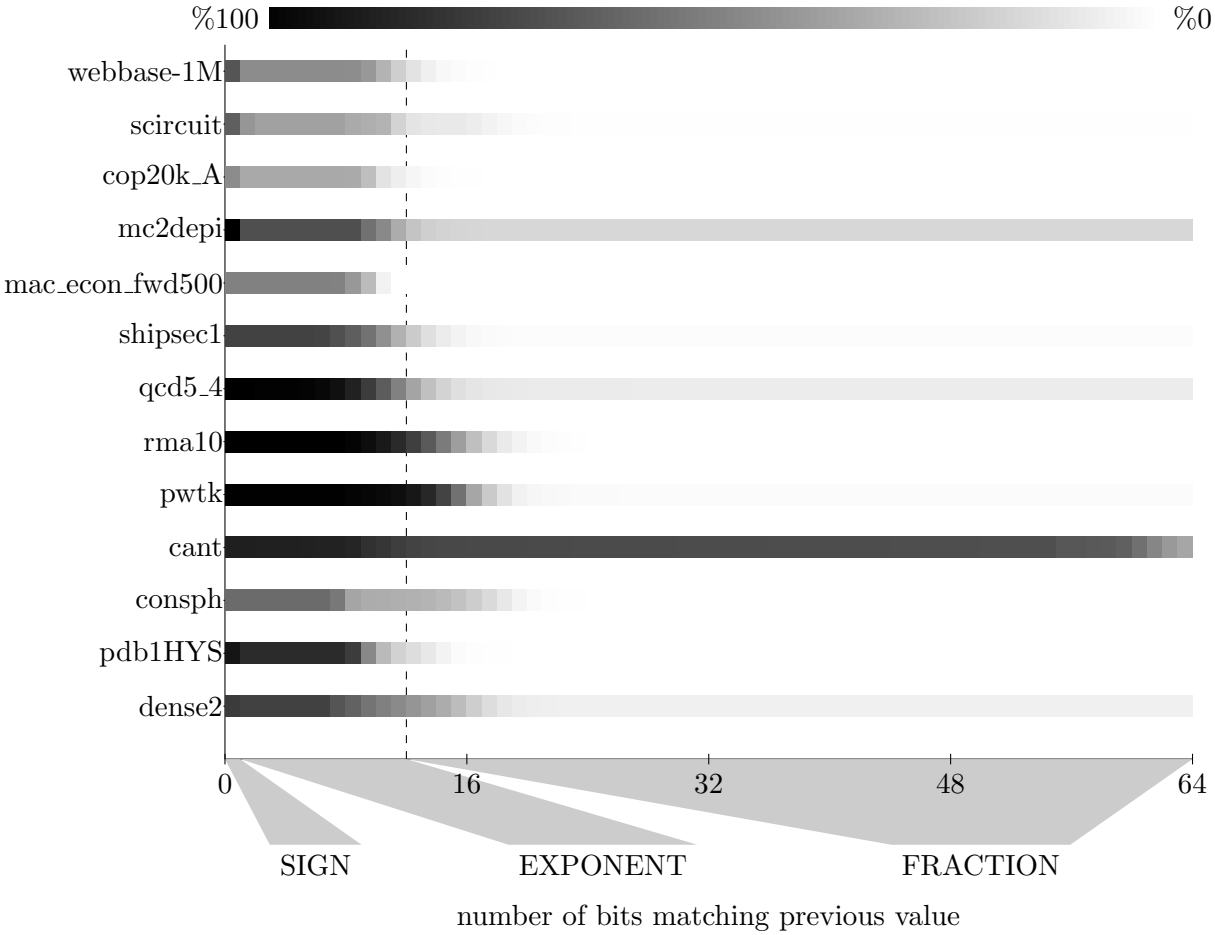


Figure 6.3: The above figure represents local prefix prediction. The figure shows the density function of 2 adjacent values sharing at least x number prefix bits. All of the data sets start at $(0, \%100)$. The curves end at the percent of values that are identical to their previous value for that dataset.

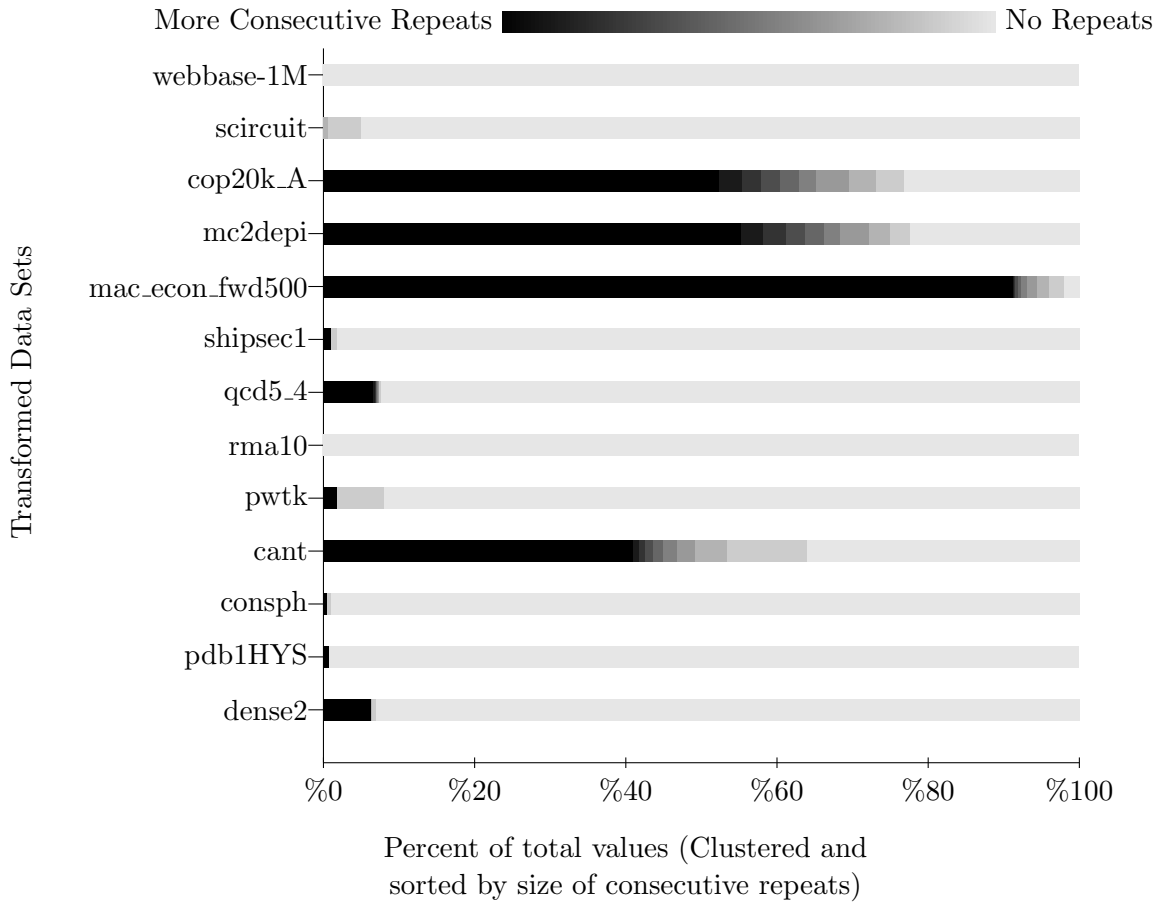
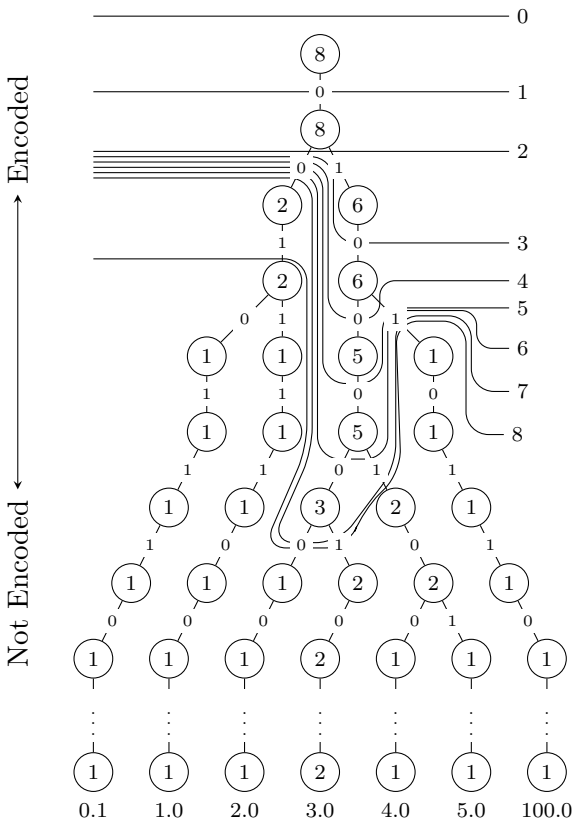
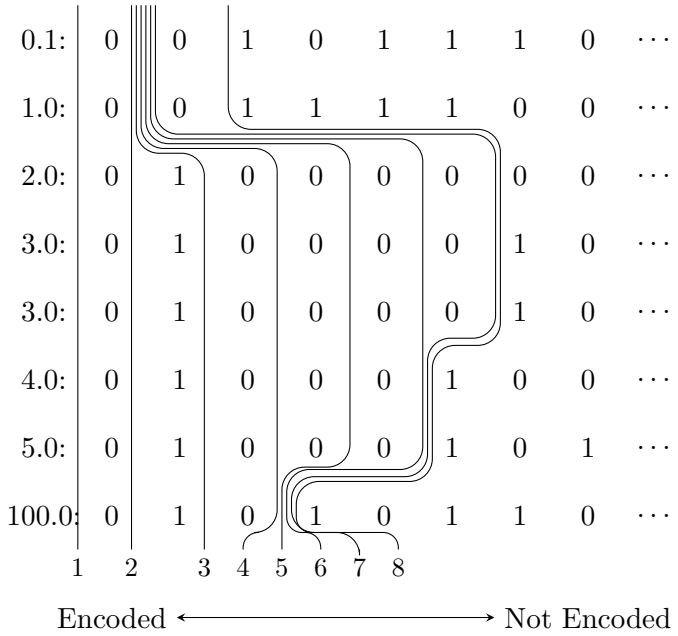


Figure 6.4: Pattern analysis using the Burrows-Wheeler Transform. Each shade represents the number of consecutive repeats in a repeating sequence. ■ represents sequences longer than 9. ■ represents sequences of length 5. ■ represents sequences equal to 1 (non-repeating).



(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 6.5: 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.

6.4.2 Prefix Compression

After BWT compression, there exists a new array of values. This array should no longer contain many long pattern sequences. However, patterns still exist among the values themselves. The values either repeat or partially repeat. Specifically, many values share common prefixes. fzip uses arithmetic codes to encode these common prefixes.

To begin with, fzip creates a large tree to represent all the values in the array. Figure **??** shows an example tree for a small dataset. The tree follows the following rules: each node has up to two children. Each edge represents a 1 bit or a 0 bit. Each node in the tree represents a prefix. The root node represents "" or no prefix. Each node also has a weight, which represents the number of values with the prefix the node represents. 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 65^{th} level (and does not contain more nodes than necessary 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 the 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: Create a pointer to a "current node" c and initiate c to the root node. Increment the weight of c (the root node). Then, with the most significant bit (the sign bit) of the floating point value, update c 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 c. This repeats until you reach the 64^{th} bit. Then, the next value gets added to the tree. This continues until the last value gets added to the tree.

fzip calculates the prefix codes by creating a partition in the tree. To start, fzip creates a partition with only the root node. Then it includes the edge with the largest cut length in the partition. This repeats until a predetermined number of edges become cut by the partition. Using a list of prefix, prefix code tuples we can represent the encoding scheme of the first 8 partitions of the example in Figure 6.5:

1. (0,)
2. (00,0), (01,1)
3. (00,0), (010,1)
4. (00,00), (0100,01), (0101,10)
5. (00,00), (01000,01), (0101,10)
6. (00,00), (010000,01), (010001,10), (0101,11)
7. (00,000), (0100000,001), (0100001,010), (010001,011), (0101,100)
8. (001,000), (0100000,001), (0100001,010), (010001,011), (0101,100)

Each added node improves the compression because of the following observation: Let the last added node equal A. The number of bits in the uncompressed (not-encoded) stream decreases by weight(A). However, the code lengths have to increase because the partition cut-size (k) increases. The code lengths equal $\log_2(k)$. So the increase in the code length equals $\log_2(k+1) - \log_2(k)$ or $\frac{1}{k}$ by using derivatives. So the codes stream will increase by $\frac{n}{k}$, where n equals to number of values in the data set. If you choose A to maximize weight(A) (a greedy algorithm) then weight(A) $>$ average edge cut $> \frac{n}{k}$. Therefore, the total size of the prefix compression, excluding overhead, keeps improving as the partition increases.

But, what if a value occurs often? Say the value 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 to variable-length arithmetic encoding.

Of course, the overhead to store 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 array size.

6.4.3 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).

To encode these repeats, we expanded the set of prefix codes. This increases the original code lengths by up to one bit. This seemed like a small trade off to make. All the repeats have the same length (64-bits) and the same code length so we can encode each as 8 bytes, instead of the 16 used for the prefixes.

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

6.5 Disscussion

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

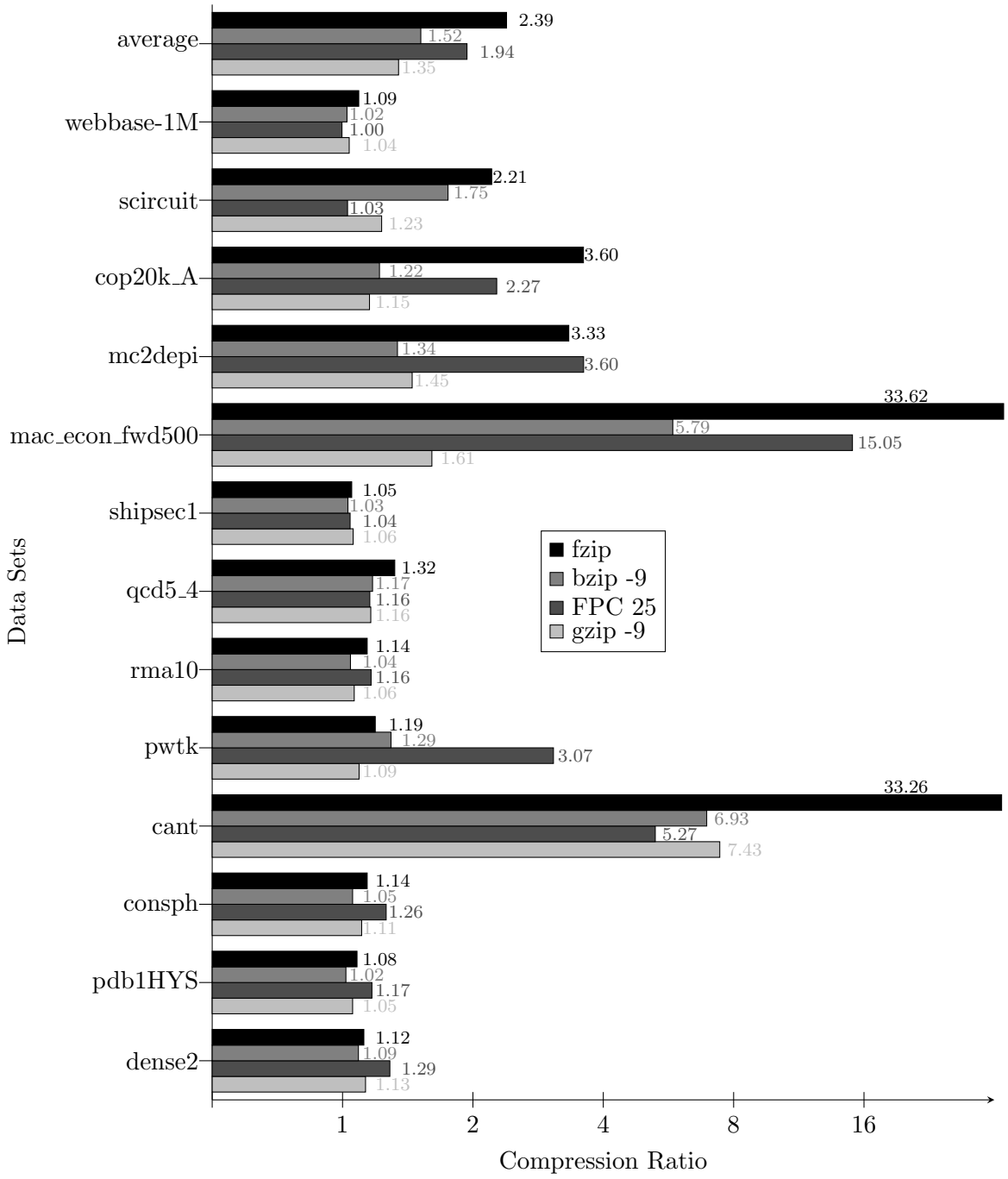


Figure 6.6: The comparison of different compression schemes shows fzip performs quite well.

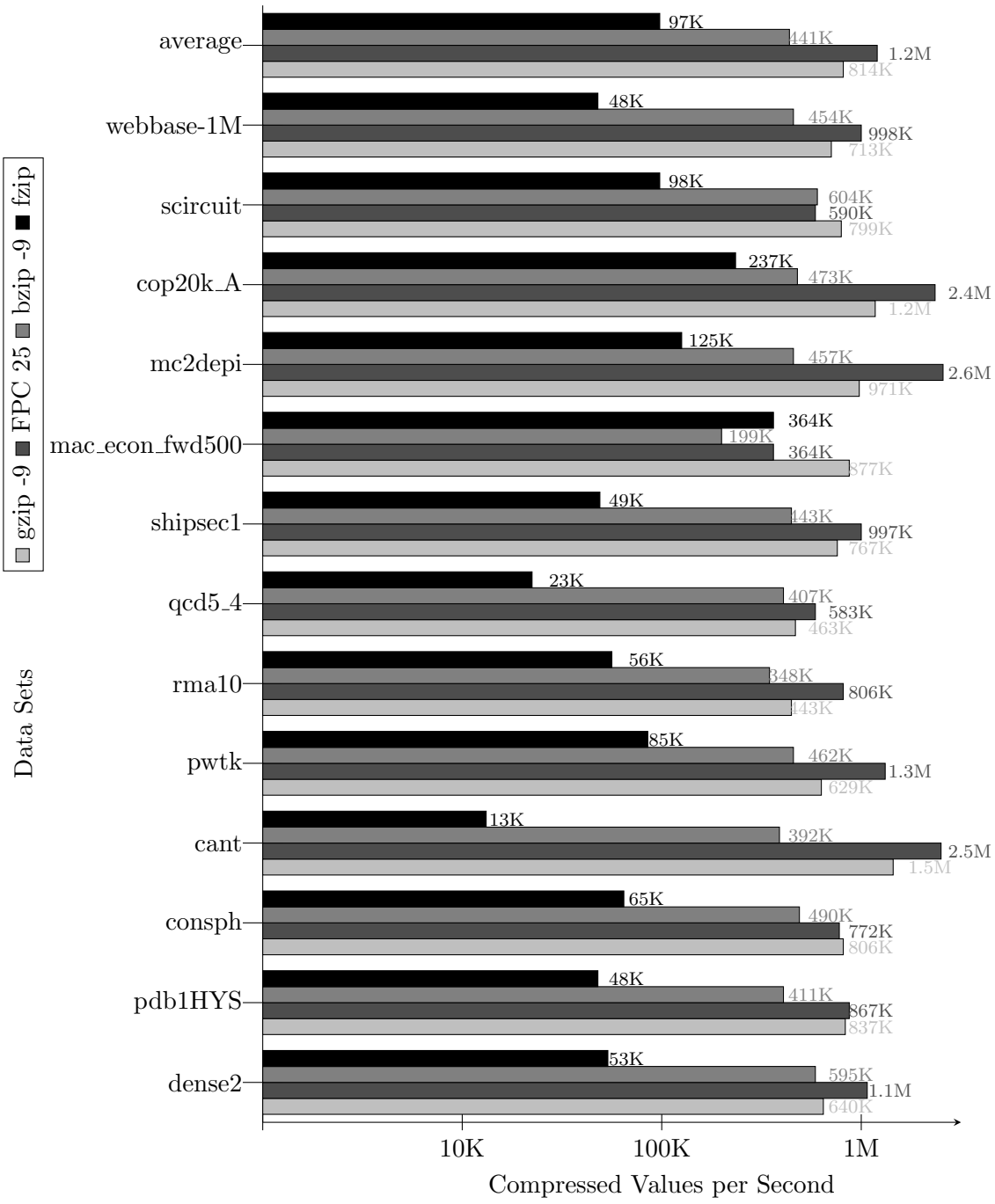


Figure 6.7: The above compression runtime analysis shows that fzip has some improvement to make to compete with other program's runtime.

CHAPTER 7. MULTI-PORT RAM

7.1 Introduction

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 achieve our desired performance.

Following Moore's Law, transistor densities on FPGAs continue to increase at an exponential rate **?**). This allows for increasingly complex System-on-Chip implementations that require high throughput communication and shared memory on networks of distributed compute nodes. ASIC designs used as accelerators in the field of high-performance computing often utilize multi-port memories for communication. By comparison, FPGA vendors do not provide scalable multi-port memories in their fabric, with most device families being limited to using dual-port memories **??**). In these cases designers must use a multi-port memory constructed with FPGA logic and RAM blocks. These soft IP cores consume significant resources if the memory must behave identically (in terms of performance) as an ASIC equivalent. As an alternative design strategy, if the multi-port memory can stall and exhibit variable multi-cycle latencies, substantially fewer resources can be used. Heavily pipelined processing elements can often tolerate these restrictions.

In this paper we present two FPGA-based multi-port memory designs that allow for scalability in terms of the number of ports as well the addressable memory space. By providing a banked RAM block architecture, our designs allow for implementations that support up to 256 ports and 1MB of memory space on current-generation FPGA devices. In contrast to previous

banked implementations in the research literature, our multi-port memories utilize buffering and reordering of memory requests. This results in throughput that approaches the ideal-case throughput, while providing an unsegmented address space. An unsegmented address space allows for simpler integration with the rest of an accelerator implementation.

The rest of this paper is organized as follows. Section 7.2 provides an overview of related work in multi-port memory design. In Section 7.3, we present our two designs (the Fully-connected and Omega memories), which provide for an explicit trade-off between implementation complexity and achievable throughput. A discussion of our evaluation methodology is provided in Section 7.4 followed by an analysis of resource usage and performance in Section 7.5. Finally, the paper is concluded with a detailed view towards planned future work in Section 7.6.

7.2 Related Work

If a multi-port memory only requires a small amount of memory space, one can synthesize the multi-port memory using only FPGA logic resources, as seen in **?**). Otherwise, soft multi-port memories utilize the dual-port RAM blocks available on most FPGAs. A review of the research literature illustrates the four design strategies using dual-port RAM blocks: *multi-pumping*, *replication*, *Live Value Table*, and *banking*.

Multi-pumping, seen in **???**), gains ports by using an internal clock and an external clock, with a clock speed of a constant multiple slower than the internal clock. This way the RAM block can process the requests of multiple ports. This approach limits the number of ports, as each added port decreases the maximum clock frequency.

Replication, seen in **???**), gains read-only ports by connecting the write ports of multiple RAM blocks together. This approach does not sacrifice clock speed or FPGA logic resources. However, each extra RAM block just provides one read-only port. Also, increasing the RAM blocks does not expand the storage space of the memory, since the data is explicitly replicated among the blocks.

Live Value Table, seen in **???**), gains ports by using a quadratically growing number of RAM blocks. This approach generates an $N \times N$ array of RAM blocks to create N ports.

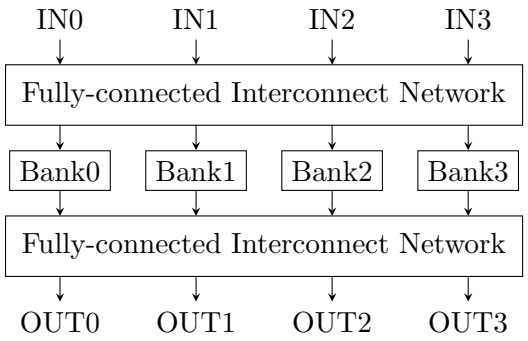


Figure 7.1: In the Fully-connected multi-port memory all the buffering occurs in the fully-connected interconnect networks.

Similar to replication, Live Value Table connects the write ports of all the RAM blocks in each row. During a write, a port writes to all the RAM blocks in one row. During a read, a port reads from all the RAM blocks in one column. One of the outputs from this column contains the most recent and the correct value for the read response. A "Live Value Table" keeps track of this information. Other techniques use an XOR operator instead of a Live Value Table **?**). These approaches have the advantage of low latency and working at relatively high frequencies. A significant drawback is the resource usage, since the number of extra RAM blocks scales quadratically with the intended memory size. Another drawback, which this approach shares with replication, is that these added RAM blocks do not expand the storage space of the memory.

Banking, seen in **???**), gains ports by adding RAM blocks to expand the memory. This allows the memory to gain a full port and increase the memory space. However, multiple ports can not access the same RAM block (and therefore the unique memory space it holds) at the same time. Also, some type of network must route signals between the ports and the banks. These previous banking approaches do not buffer requests and therefore restrict the access of each port to a fraction of the memory space, explicitly segmenting each bank. By comparison, the work presented in this paper buffers and reorders messages, allowing unsegmented access to memory.

7.3 Architecture

As previously mentioned we implemented two different memory designs, hereafter referred to as the *Fully-connected* and *Omega* multi-port memories. As will be further demonstrated in Section 7.5, a key differentiator is that the Fully-connected memory has better throughput, while the Omega memory scales better in terms of resource usage. However, both memories share common characteristics. Both designs use single-port RAM blocks for the memory banks, and utilize reorder queues. Also, both utilize a network structure for routing between ports and banks, and buffer requests to resolve contention.

7.3.1 Fully-connected Multi-port Memory

The Fully-connected multi-port memory (Figure 7.1) uses fully-connected interconnect networks. The first fully-connected network routes requests from the input ports to the banks. The second routes read responses from the banks to the output ports.

7.3.1.1 Memory Banks

For any banking approach, a memory with N ports requires at least N RAM blocks. Each RAM block holds a unique segment of the total memory space. We have multiple options to decide how to segment the memory space. The simplest option assigns the first N^{th} of the address space to Bank0, the next N^{th} to Bank1, and so on. However, this approach can easily cause bottlenecks. For example, assume all the processing elements start to read from a low address located in Bank0 and continue to sequentially increment the read addresses. All the requests would route to Bank0, necessitating multiple stalls. The interleaving memory address space that our design uses decreases the chance these specific types of bottlenecks occur.

7.3.1.2 Fully-connected interconnect network

A fully-connected interconnect network (Fig. 7.2) consists of multiple arbiters. An arbiter routes data from several inputs to one output, and typically consist of simple FIFOs, a multiplexer, and some control logic. In Fig. 7.2, FIFOs A0 to A3 and MUX A construct one of the

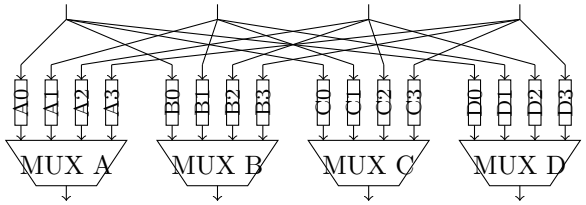


Figure 7.2: Fully-connected interconnect network

four arbiters needed for a four-port fully-connected interconnect network.

We use a simple round robin scheme to resolve contention. Assuming the arbiter most recently read from FIFO A0, the arbiter would continue to read from FIFO A0 until it is empty. Then, the arbiter reads from FIFO A1. Again, the arbiter continues to read from FIFO A1 until it is empty, and the cycle repeats. This control scheme has the advantage that it achieves high throughput and requires little control logic.

Generally, a N-by-N fully-connected interconnect network consists of N, N-to-1 arbiters. The arbiters act independently. This independence allows for good arbitration schemes that achieve high throughput and low latency. Unfortunately, as the size of a fully-connected interconnect network grows, the more space the FIFOs and multiplexers require. A 8-to-1 multiplexer requires approximately twice the number of resources of a 4-to-1 multiplexer. This means the area the multiplexers require grows by around N^2. The number of FIFOs grows by N^2 as well.

7.3.2 Omega Multi-port Memory

The Omega multi-port memory (Figure 7.3) has hardware structures designed for scaling. Instead of using fully connected interconnect networks, area efficient multi-stage interconnect networks (MIN) route signals to and from the memory banks. In addition, this memory uses N linked list FIFOs to buffer incoming requests, instead of N^2 FIFOs. These two structures pair well, because they both save logic resources. However, both share a common restriction; neither structure can simultaneously send multiple buffered messages, from the same port, to different banks.

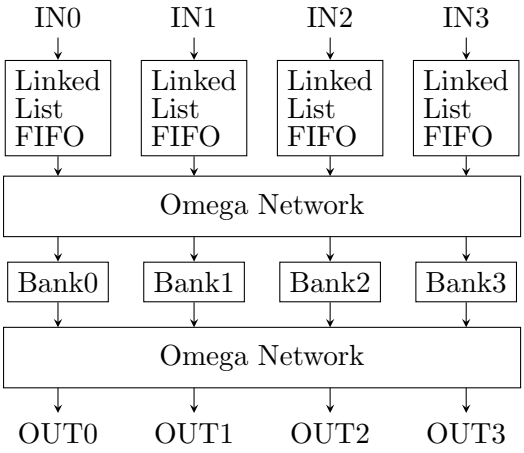


Figure 7.3: In the Omega multi-port memory all the buffering occurs in the linked list FIFOs. The use of multi-stage interconnect networks, in this case Omega networks, helps reduce the area of the design.

7.3.2.1 Omega Network

An Omega network consists of columns of Banyan switches **??**). A Banyan switch synthesizes to two multiplexers. In the ON state, the switch crosses data over to the opposite output port. As an illustrative example, the second column in Fig. 7.4 only contains switches in the ON state. In the OFF state, the switch passes data straight to the corresponding output port. The first and last columns in Fig. 7.4 only contain switches in the OFF state.

The Omega network has features that make it attractive in a multi-port memory design. If we switch whole columns of Banyan switches ON or OFF, we can easily determine where signals route by XORing the starting port index with the bits controlling the columns. For example, in Fig. 7.4, the control bits equal 010_2 or 2 and input port 2 (010_2) routes to output port 0. Not coincidentally, the same configuration routes in reverse. Input port 0 routes to output port 2 and input port 2 routes to output port 0. This means the design can use identical control bits for both the receiving and sending Omega networks.

In the Omega multi-port memory design, the control for this network increments every clock cycle. As an example, input port 5 would connect to output port 5, then port 4, 7, 6, 1, etc., until it cycles around again. This means each input connects to each output an equal number of times.

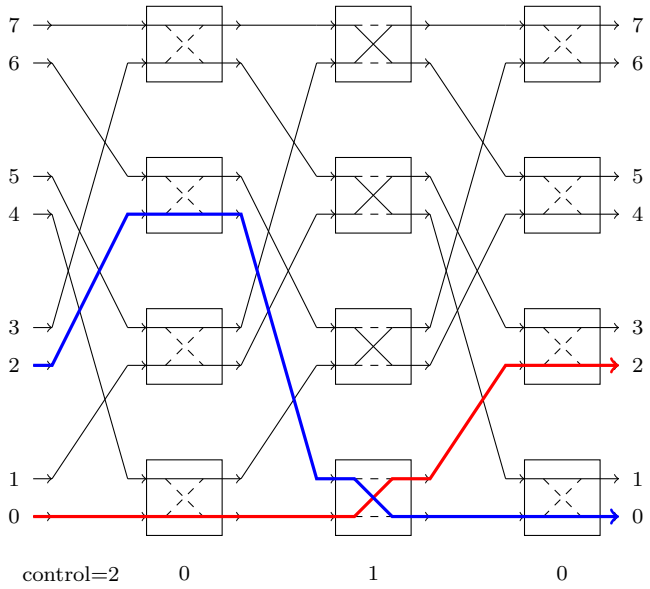


Figure 7.4: An 8-by-8 Omega network. We turn columns on or off to rotate between different routing configurations.

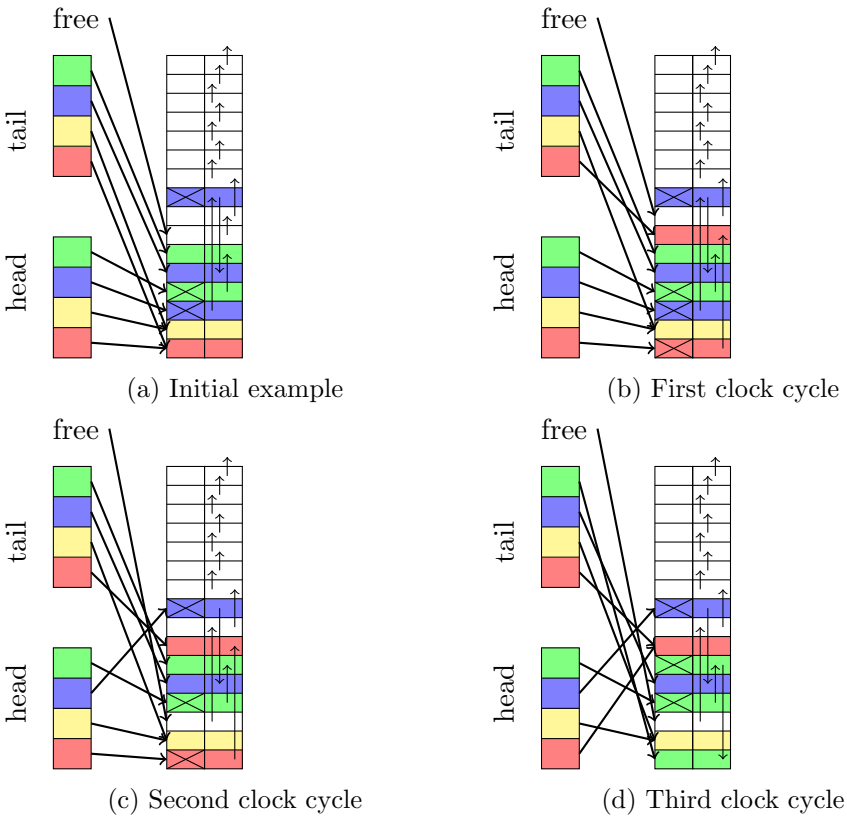
7.3.2.2 Linked List FIFO

The partnering hardware structure, the linked list FIFO (Fig. 7.5), contains several internal FIFOs with no predefined space in a single RAM. Similar to a software linked list, there exists a free pointer that points to the beginning of the free space linked list. Linked list FIFOs have previously seen use in multi-core CPU **?**) and FPGA **?**) designs.

Due to the linking pointers, the size of the RAM now needs $O(NlogN)$ space to store N elements. However $logN$ grows slowly. For example, data stored in a 64-bit wide by 1024 deep RAM would need an additional 11-bit wide by 1024 deep RAM for the linking pointers. An illustrative example of the linked list FIFO is shown in in Fig. 7.5, which uses a 16 deep RAM and 4 FIFOs.

In the initial state (Figure 7.5a), the red and yellow FIFOs have no messages. The blue FIFO has two messages. And, the green FIFO has one message. However, every FIFO reserves one space for the next incoming value. This limits the total available space in the linked list FIFO to $TOTAL\_DEPTH - FIFO\_COUNT$.

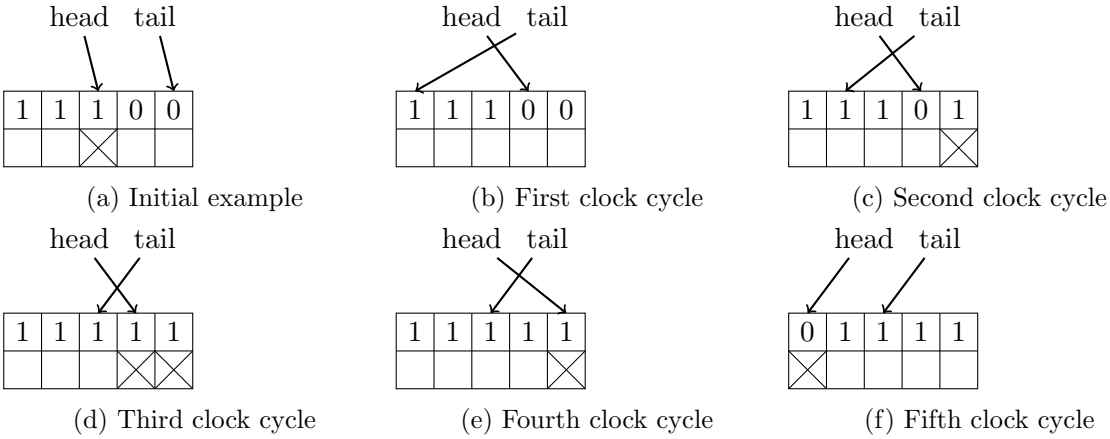
- On the first clock cycle (Fig. 7.5b), the linked list FIFO receives a push containing a red



(a) Initial example (b) First clock cycle

(c) Second clock cycle (d) Third clock cycle

Figure 7.5: A linked list FIFO during 3 clock cycles of operation



(a) Initial example (b) First clock cycle (c) Second clock cycle

(d) Third clock cycle (e) Fourth clock cycle (f) Fifth clock cycle

Figure 7.6: Reorder queue example.

message. The new red message gets stored in the reserved space at the tail of the red linked list. The free linked list pops one space. That space gets pushed on to the red linked list.

- On the second clock cycle (Fig. 7.5c), the linked list FIFO receives a pop for a blue message. A blue message gets popped from the head of the blue linked list. The newly freed space gets pushed on to the free linked list.

- On the third clock cycle (Fig. 7.5d), the linked list receives a pop for a red message and a push for a green message. In this case the space that the red message was popped from gets pushed onto the green linked list. The free space linked list stays the same.

7.3.3 Reorder Queue

The buffering in both designs ensures relatively high throughput, however, this buffering causes a problem for both memories, as read responses from different banks from the same port may come back out of order. Although out of order reads do not always cause an issue, to alleviate this issue we add reorder queues (Fig. 7.7) to both multi-port memory designs.

A reorder queue behaves similarly to a FIFO. However, some of the values in between the head and tail pointer exist "in flight" and not at the reorder queue memory. The reorder queue keeps track of the presence of messages with a bit array (a 1-bit wide RAM).

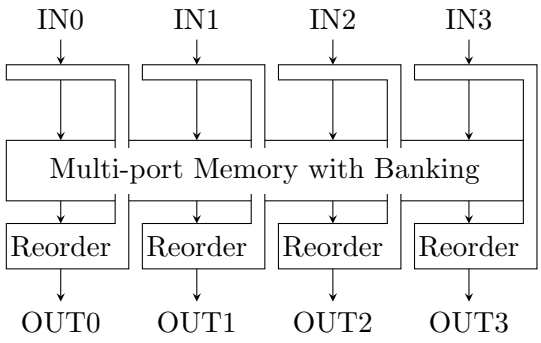


Figure 7.7: A reorder queue tags incoming read requests with an ID that allows the reorder queue to know the correct order of the read responses.

Figure 7.6 shows an example with 5 clock cycles of operation. In the initial state (Fig. 7.6a), the reorder queue has one present message and one in flight message.

- On the first clock cycle (Fig. 7.6b), the present message at the head gets popped from the queue. A new message increments the tail, but the message remains in flight until it arrives at the reorder queue.
- On the second clock cycle (Fig. 7.6c), a new message arrives at the reorder queue, however, it does not arrive at the head of the queue so no message can get popped.
- On the third clock cycle (Fig. 7.6d), a message arrives at the head of the reorder queue.
- On the fourth clock cycle (Fig. 7.6e), this message at the head of the reorder queue gets popped. If the reorder queue did not exist, the message that appeared on clock cycle 2 would have reached the output first even though it was sent later.
- On the fifth clock cycle (Figure 7.6f), a message arrives. However, the meaning of 1 or 0 in the 1-bit RAM switched after the pointers wrapped around the end of the RAM. Instead of 1 meaning present, 1 now means in flight. This semantic flipping allows the use of only one write port on the 1-bit RAM (instead of two if the bits flipped after popping a message), saving on memory-related resources.

7.4 Evaluation Methodology

We implemented a small resource and a large resource version of each memory. The small version does not use RAM blocks for buffering, and limits linked list FIFOs and reorder queues to a depth of 64. The large version does use RAM blocks for buffering and limits these memories to a depth of 512. However, in both cases we limit the depth of the FIFOs in the fully-connected interconnect network to 32 since the number of FIFOs in it grows by $O(N^2)$.

For synthesis we used Xilinx Synthesis Tools (XST) and targeted the Xilinx Virtex-7 V2000T. The V2000T has a relatively large number of logic blocks and RAM blocks, which helps us push the limits of these designs. These designs should also work well on Altera FPGAs, given the commonalities between the devices. Both Xilinx and Altera use a base depth of 32 for distributed RAM and a base depth of 512 for RAM blocks.

We used the ModelSim logic simulator to evaluate the performance of each configuration. The testbench used for evaluation consists of four benchmarks. Each benchmark tests the read performance of different memory access patterns: *sequential*, *random*, *congested*, or *segregated*.

The *sequential* benchmark begins by sending a read request to memory address 0 on each port. On the next clock cycle each port requests data from memory address 1. This continues unless a port stalls. On a stall, the memory address of that port does not increment until the memory resolves the stall.

The *random* benchmark begins by sending a read request to a random memory address on each port. On the next clock cycle, every port gets a new random memory address to read from. This continues until the end of the benchmark.

The *congested* benchmark begins by sending a read request to memory address 0 on each port. On subsequent clock cycles, the memory address does not change. This results in all the ports attempting to access the same memory bank. The purpose of this benchmark is to demonstrate the worst-case performance for any type of multi-port memory with banking, including our designs.

The *segregated* benchmark begins by sending a read request to memory address i on each port, where i equals the index of the requested port. This address does not change on subsequent

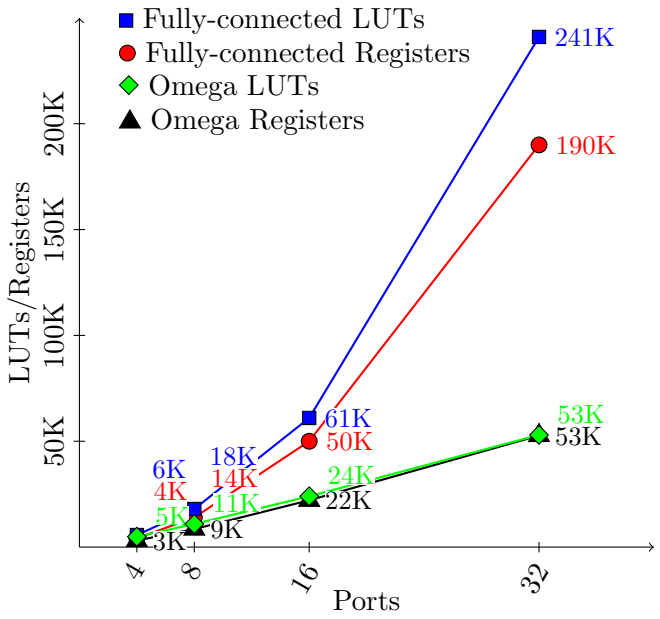


Figure 7.8: The effect of varying the number of ports on FPGA resource utilization (area). The Fully-connected memory grows by approximately N^2 and the Omega memory grows almost linearly.

clock cycles. This results in all the banks receiving an equal number of requests. However, unlike the sequential and random benchmarks, there exists an uneven distribution of requests among all N^2 port to bank connections.

We calculate the throughput of a given benchmark by measuring the ratio of read requests to potential read requests. If no stalls occur, the throughput equals 100%. We calculate the latency by measuring the number of clock cycles between the last read request and the last read response.

7.5 Results and Analysis

We present most of our results of different memory configurations in Table 7.1. From a quick analysis of the results we can confirm several expected outcomes. The Omega memory uses fewer FPGA resources than the Fully-connected memory, particularly for memories with more ports. The Fully-connected memory achieves better throughput particularly for the segregated benchmark. We further analyze the effect of varying the number of ports, the depth of the

Table 7.1: Analysis of the two multi-port memory designs.

	Ports		4	8	16	32	64	128	256
	Memory Space		16KB	32KB	64KB	128KB	256KB	512KB	1M
Linked List FIFO and Reorder Queue Depth set to 64		Fully-connected Multi-port Memory							
	Resource Utilization Virtex 7 V2000T[2]	Registers	4K	14K	50K	190K	728K		
		LUTs	5.7K	18K	61K	241K	906K		
		BlockRAM	4	8	16	32	64		
		Clock frequency	345Mhz	313Mhz	256Mhz	273Mhz	230Mhz		
	Sequential	Throughput	100%	100%	100%	100%	50%		
		Latency [1]	16	20	36	64	128		
	Random	Throughput	97%	93%	88%	72%	49%		
		Latency [1]	66	65	85	97	144		
	Congested	Throughput	25%	13%	6%	3%	2%		
		Latency [1]	105	230	490	1034	2780		
	Segregated	Throughput	100%	100%	100%	100%	100%		
		Latency [1]	16	24	34	63	61		
		Omega Multi-port Memory							
	Resource Utilization Virtex 7 V2000T[2]	Registers	3K	9K	22K	53K			
		LUTs	5K	11K	24K	53K			
		BlockRAM	4	8	16	32			
		Clock frequency	258Mhz	257Mhz	260Mhz	262Mhz			
	Sequential	Throughput	100%	100%	100%	100%			
		Latency [1]	17	25	37	56			
	Random	Throughput	94%	83%	68%	52%			
		Latency [1]	72	110	131	193			
	Congested	Throughput	25%	13%	6%	3%			
		Latency [1]	250	462	786	1046			
	Segregated	Throughput	25%	13%	6%	3%			
		Latency [1]	247	461	756	1043			
Linked List FIFO and Reorder Queue Depth set to 512		Fully-connected Multi-port Memory							
	Resource Utilization Virtex 7 V2000T[2]	Registers	4.2K	14K	50K	191K	744K		
		LUTs	5.3K	17K	60K	241K	928K		
		BlockRAM	7	13	25	48	96		
		Clock frequency	352Mhz	315Mhz	253Mhz	271Mhz	230Mhz		
	Sequential	Throughput	100%	100%	100%	100%	100%		
		Latency [1]	16	20	36	68	100		
	Random	Throughput	96%	95%	94%	99%	98%		
		Latency [1]	134	296	512	553	616		
	Congested	Throughput	25%	13%	6%	3%	2%		
		Latency [1]	105	231	491	1019	2750		
	Segregated	Throughput	100%	100%	100%	100%	100%		
		Latency [1]	16	23	38	61	63		
		Omega Multi-port Memory							
	Resource Utilization Virtex 7 V2000T[2]	Registers	3.0K	8.4K	23K	53K	125K	300K	677
		LUTs	7.3K	16K	36K	77K	163K	345K	746
		BlockRAM	9	17	32	65	129	257	513
		Clock frequency	234Mhz	234Mhz	230Mhz	230Mhz	225Mhz	202Mhz	175
	Sequential	Throughput	100%	100%	100%	100%	100%	100%	100

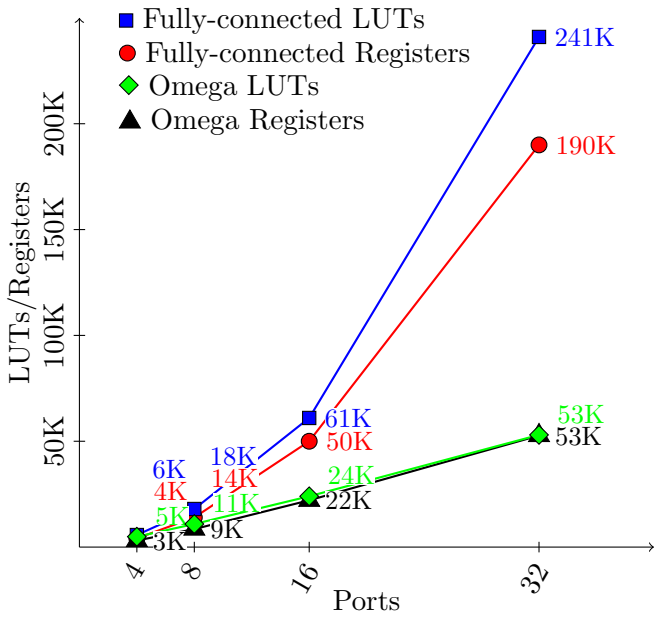


Figure 7.9: The effect of varying the number of ports on throughput of the random memory access benchmark on the small resource memories.

buffering structures, and the bit width of the data.

7.5.1 Varying the number of ports

In terms of area, Figure 7.8 shows the effect on FPGA logic resources due to varying the number of ports. As expected, the Fully-connected memory consumes resources at a rate of approximately $O(N^2)$. The Omega memory consumes resources at a slower rate of approximately $O(NlogN)$. At 8 ports, the Fully-connected memory consumes 50% more resources than the Omega memory, and either design is viable on the target device. However, as the number of ports are increased, the Fully-connected memory runs out of resources quicker than the Omega memory. Table 7.1 grays the cells reporting the performance of the Fully-connected memory with 128 and 256 ports, because the Fully-connected memory for these configurations uses more than the available amount of resources on the Virtex-7 V2000T. The Omega memory does not reach this limit until it has more than 256 ports.

In terms of performance, Figure 7.9 shows that increasing the number of ports decreases throughput, and Table 7.1 shows increasing the number of ports increases latency. As expected,

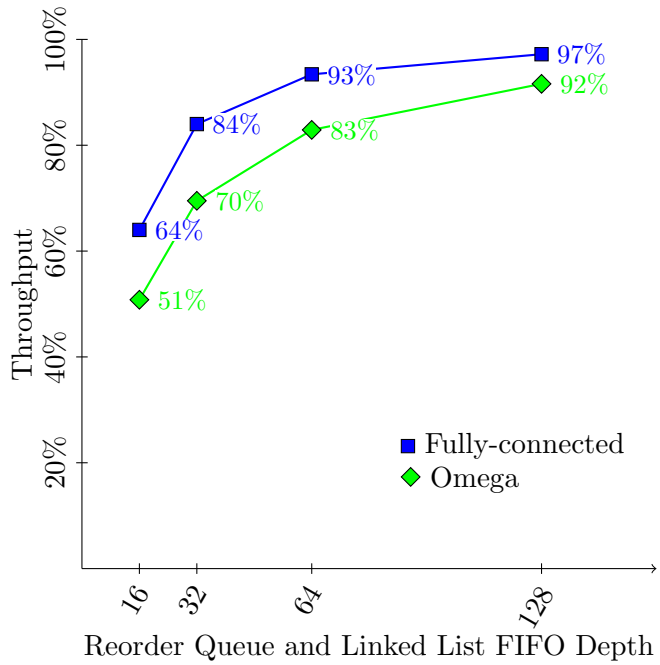


Figure 7.10: The effect of varying the depth of the linked list FIFOs and reorder queues on throughput of the random memory access benchmark (using 8-port memories).

throughput decreases a little faster for the Omega memory. In both memories the latency grows almost linearly with the number of ports, because of the round robin contention resolution scheme in both. On average it takes $\frac{N}{2}$ clock cycles to start processing the first memory request.

7.5.2 Varying the buffer depth

Increasing the buffer depth, i.e. the reorder queue depth and the linked list FIFO depth, increases the throughput of the memories. Figure 7.10 shows that the throughput increases by around $O(1-(\frac{p-1}{p})^N)$, where p equals the number of ports and N equals the buffer depth. $1-(\frac{p-1}{p})^N$ equals the probability that at least one of the last N memory requests requested data on bank0 (or any specific bank). This approximately equals the probability that the next FIFO in the round robin has at least one message.

The buffer depth imposes a hard limit on the number of ports the Omega memory has. The Omega memory must have a buffer depth larger than the number of ports. Table 7.1 grays

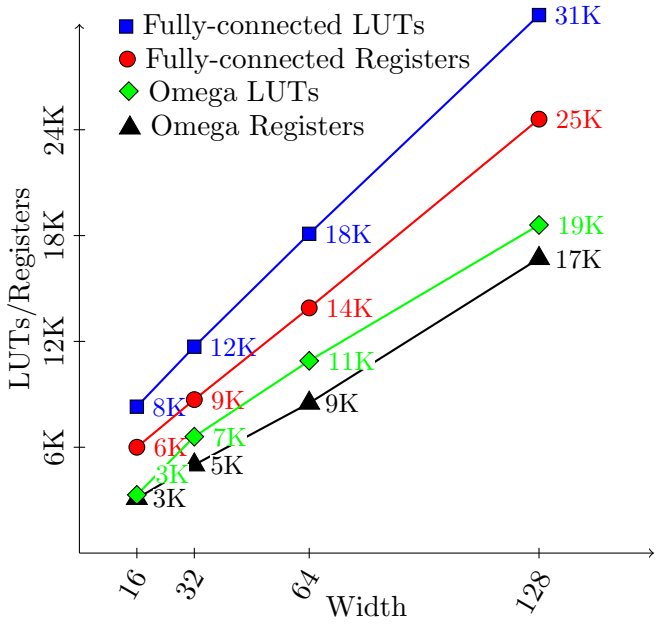


Figure 7.11: The effect of varying the bit-width of the memory on FPGA resource utilization.

out the cells reporting the performance of the small resource Omega multi-port memory, which have 64, 128, or 256 ports, because the linked list FIFOs have a depth of 64.

The Fully-connected memory does not share this restriction, however, an interesting anomaly occurs in the same situation. Table 7.1 shows that the sequential throughput of the small resource Fully-connected memory with 64 ports equals 50%. In this case, the messages wait 64 clock cycles to pass through the first interconnect network and then another 64 clock cycles to pass through the second. This happens because messages keep arriving at the FIFO the arbiter had just finished processing, similar to arriving at a bus stop just as the bus leaves. Since the reorder queue only allows 64 in flight messages, the 128 clock cycle latency only allows for 50% throughput. The problem disappears when the reorder queue is of size 128 or greater.

Increasing the buffer depth increases the latency. The buffers fill up over time as they attempt to prevent the memory from stalling. Full buffers means latency increases by the depth of the buffer. So in benchmarks with contention, the latency increases linearly with the buffer depth.

Increasing the buffer depth increases FPGA utilization. The increase in buffer depth affects the Omega memory more since the Fully-connected memory does not have linked list FIFOs.

If we use RAM blocks for buffers, any depth less than 512 results in using approximately the same number of resources. Consequently in this configuration we only implement FIFO depths of 64 and 512. Unsurprisingly, if buffers consist entirely of distributed RAMs (LUT resources), FPGA utilization increases linearly with the buffer depth.

7.5.3 Varying the data bit width

Data width only effects resource utilization. As Fig. 7.11 shows, FPGA utilization scales linearly with the data bit width. However, bit width does effect throughput when measuring by bytes per second instead of by percentage. The bytes per second measurement equals $PERCENT\_THROUGHPUT \times PORT\_COUNT \times BIT\_WIDTH \times CLOCK\_FREQUENCY$. For example, the throughput on the random benchmark of the Omega memory with 256 ports is 172 GB/s.

7.6 Conclusions and Future Work

In total, the contributions of this work include the design of two multi-port memories with banking and the components used to construct them. Based on the analysis of the results we recommend the Fully-connected multi-port memory for designs that require low latency and high throughput. However, if the design requires 16 or more ports we recommend the Omega multi-port memory. In other cases, either version should work fine.

Overall, we find that multi-port memories with banking can provide high throughput communication to distributed compute nodes. We also find that these memories provide a substantial amount of shared memory. Upon the time of publication we will provide the source code for the memory designs on our research group's Github page, so that other researchers can further analyze our work and integrate these cores into future high-performance reconfigurable computing applications.

In terms of planned future work, we are currently considering three main ideas: more aggressive pipelining, better contention resolution, and use of dual-port RAM blocks.

7.6.1 Improvements in pipelining

Currently most of our design configurations achieve an estimated clock frequency between 200Mhz and 300Mhz. The Virtex 7 V2000T (with -2 speed grade) has a maximum achievable frequency of approximately 500Mhz. Going through the design and pipelining the critical paths should increase the maximum frequency by 100Mhz.

7.6.2 Better contention resolution

Currently the arbiters in the Fully-connected multi-port memory process the FIFOs in a round robin scheme. In this scheme, processing empty FIFOs still consumes one clock cycle. Skipping empty FIFOs would improve latency and performance.

Improving the contention resolution of the Omega memory has more complications. The Omega network limits the number of routes from input ports to the banks. However, we believe improvements exist.

7.6.3 Use of dual-port RAM blocks

In either version (Fully-connected or Omega), designing an N-by-N multi-port memory as two $\frac{N}{2}$-by-$\frac{N}{2}$ multi-port memories with banks connected by dual-port RAM blocks, instead of using single-port RAM blocks, can achieve superior results. The matching memory banks (by their index) of the two smaller designs would combine to make dual-port RAM blocks. This makes the estimated throughput and latency equal to the memory with half the number of ports in Table 7.1. This also makes the estimated resource utilization twice the resource utilization of the memory with half the number of ports in Table 7.1. For example, a small resource Fully-connected multi-port memory with 32 ports utilizing dual-port RAM blocks would use around 100K registers, 122K LUTs and 32 RAM blocks, verses 190K registers, 241K LUTs and 32 RAM blocks when only utilizing single-port RAM blocks.

CHAPTER 8. CONCLUSIONS

This is a lot of work for just a FPGA based SpMV implementation. However we believe SpMV is an important computation kernel and that FPGAs can outperform CPUs and GPUs. As mentioned SpMV with large matricies (¿ 1 billion values) perform poorly on CPUs because of cache issues and do not fit in the memory of GPU cards. These large matrix applications is where we expect FPGA platforms will be used for SpMV calculations.

8.1 EXPECTED RESULTS

R^3, our previous work, achieved an average performance of 6 GFLOPS on the dataset used by). We expect to be able to double this performance to an average of 12 GFLOPS.

8.2 Timeline

I have targets for the components that need to be made to create our second generation of R^3. TODO: table

Table 8.1: Timeline

scratchpad	March 1
fzip (floating point compression)	April 1
matrix compression	May 1
Multiply Accumulator	June 1
Matrix hardware decoder	July 1
Memory Controller	August 1
R^3 gen. 2	September 1