Sparse Matrix Vector Multiplication on FPGA-based Platforms

by

Kevin R. Townsend

A dissertation 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

This dissertation implements a sparse matrix vector multiplication (SpMV) algorithm for FPGAs. CPUs, GPUs and FPGAs are processors that run different algorithms at different speeds. Much like how the performance of an athlete depends on the design of the course as much as it does on the particular athlete. SpMV is an interesting course in that tweaking the size and sparsity of the matrix will change who wins the race. For example, CPUs compute small matrices quickly due to the fact the matrix and vector can fit in on-chip cache. GPUs compute structured matrices quickly due to how they preprocess the matrix. Currently FPGAs compute matrices at a relatively consistent speed, but we show compressible matrices can achieve better performance. In this paper we implement an FPGA-based SpMV algorithm and use features of the course (matrix) to run faster.

CHAPTER 1. INTRODUCTION

This dissertation outlines a method to achieve high performance sparse matrix vector multiplication (SpMV) on FPGA platforms.

People use SpMV in a variety of applications including information retrieval [Page et al. (1999)], text classification [Townsend et al. (2014)], and image processing [Wang et al. (2011)]. Often the SpMV operations are iterative or repetitive 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 compute 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 [Davis and Chung (2012)]. The rock being CPUs that compute SpMV on matrices with less than 10 million values well. The hard place being FPGAs that compute SpMV on matrices with more than 400 million values well (or at least not as badly as CPUs and GPUs).

In the Chapter 2, we describe the previous approaches to SpMV on CPUs, GPUs and

FPGAs. In Chapters 3, 4, 5, 6, and 7, we discuss our optimizations for FPGAs. In Chapter 8 we present our results. In Chapter 9 we conclude the paper.

CHAPTER 2. BACKGROUND

In its simplest form sparse matrix vector multiplication is the operation $y = Ax$, where A is an $M \times N$, x is a vector of length N, 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, or people you do not know. 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 of matrices 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 Format (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 format (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, 3, 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 on its 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 required 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 significant if nnz/M (number of non-zero elements per row) is small.

2.2 CPU

The sparsity of the matrix causes CPUs to perform below their potential. Recent Intel publications show an average performance of 408 GFLOPS for matrix matrix multiplication and 100 GFLOPS for matrix vector multiplication but publishes an average performance of 17 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 400 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.

There exists several optimizations to improve the performance of SpMV. 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 Chapter 3, which talks about our approach to computing SpMV on FPGAs.

2.3 Compressed Sparse Row Format (CSR)

The first SpMV optimization, compressed sparse row (CSR), is the simplest. The optimization compresses the row indices. 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. CSR format stores the traversal index of 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. This change saves up to 4 bytes per element or 25% over COO format. The CSR format of the matrix 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 hybrid formats. We have pessimistic view of hybrid formats, because this results in performing SpMV on 2 matrices, both of which are sparser than the original. In general, 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 simplified example does not actually save space because of the extra zeros 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, introduced later in 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

Before discussing storage formats specific to GPUs, it is important to understand GPUs play the computation game differently than 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 GPU supports up to 30720 threads whereas the CPU supports 30 threads.

When using a COO format based GPU implementation each thread processes $nnz/30720$ values. Some synchronization occurs to ensure the correct y values are stored. This implementation 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 Bell and Garland (2008) introduced ELLPACK, 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.

$$\begin{array}{c}\text{Matrix}\\\text{Data}\end{array} = \begin{bmatrix} A_{11} & A_{14} & A_{17} & 0 \\ A_{25} & A_{28} & 0 & 0 \\ A_{32} & A_{33} & A_{36} & A_{37} \\ A_{41} & A_{45} & 0 & 0 \\ A_{53} & A_{54} & A_{57} & A_{58} \\ A_{62} & A_{65} & 0 & 0 \\ A_{72} & A_{73} & A_{76} & A_{78} \\ A_{83} & A_{84} & A_{85} & A_{86} \end{bmatrix}, \begin{array}{c}\text{Column}\\\text{Indices}\end{array} = \begin{bmatrix} 0 & 3 & 6 & \backslash 0 \\ 4 & 7 & \backslash 0 & \backslash 0 \\ 1 & 2 & 5 & 6 \\ 0 & 4 & \backslash 0 & \backslash 0 \\ 2 & 3 & 6 & 7 \\ 1 & 4 & \backslash 0 & \backslash 0 \\ 1 & 2 & 5 & 7 \\ 2 & 3 & 4 & 5 \end{bmatrix} \tag{2.2}$$

Like CSR each thread computes one row of the matrix. However, the ELLPACK matrix is stored in column major order. This enables coalesing memory access. Coalesing memory access essentially means different threads are accessing the same cache lines. The ELLPACK format for the example would be:

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

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

The authors also deal with abnormally large rows by creating a hybrid format and store the values of rows with too many into a second COO matrix.

2.8 Block-ELLPACK

ELLPACK has seen a lot of variations in the research literature. We discuss one design that marries BSR with ELLPACK called BELLPACK [Choi et al. (2010)]. We simplify the design a little here. The idea is to combine the index compression of BSR with the memory coalesing benefit of ELLPACK. In this design, we take the 2 densest sub-blocks in every set of 2 rows. The Block-ELLPACK format for the example in equation 2.1 is shown below:

$$\begin{matrix}\text{Matrix}\\ \text{Data}\end{matrix} = \begin{bmatrix} \begin{bmatrix} A_{14} & 0 \\ 0 & A_{25} \end{bmatrix} & \begin{bmatrix} A_{17} & 0 \\ 0 & A_{28} \end{bmatrix} \\ \begin{bmatrix} 0 & A_{32} \\ A_{41} & 0 \end{bmatrix} & \begin{bmatrix} 0 & A_{36} \\ A_{45} & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & A_{53} \\ A_{62} & 0 \end{bmatrix} & \begin{bmatrix} A_{54} & 0 \\ 0 & A_{65} \end{bmatrix} \\ \begin{bmatrix} A_{72} & A_{73} \\ 0 & A_{83} \end{bmatrix} & \begin{bmatrix} 0 & A_{76} \\ A_{85} & A_{86} \end{bmatrix} \end{bmatrix}, \begin{matrix}\text{Column}\\ \text{Indices}\end{matrix} = \begin{bmatrix} 3 & 6 \\ 0 & 4 \\ 1 & 3 \\ 1 & 4 \end{bmatrix} \tag{2.3}$$

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

VALUE: A_{14}, 0, 0, A_{41}, 0, A_{62}, A_{72}, 0, 0, A_{25}, A_{32}, 0, A_{53}, 0, A_{73}, A_{83}, A_{17}, 0, 0, A_{45}, A_{54}, 0, 0, A_{85}, 0, A_{28}, A_{36}, 0, 0, A_{65}, A_{76}, 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 performance one can be calculated by creating a matrix matrix multiplication engine to load on the FPGA. The work in Cappello and Strenski (2013) provided 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, and we

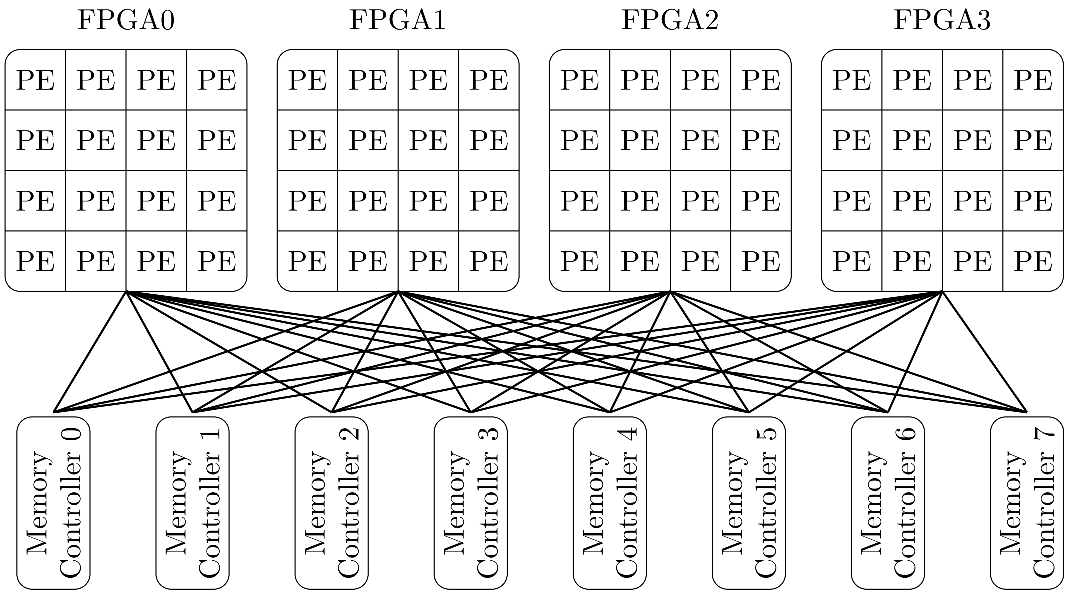


Figure 2.1: R^3 implementation on the Convey HC-2 coprocessor: 4 Virtex-5 LX330 FPGAs tiled with 16 R^3 SpMV processing elements (PE) each. Each Virtex-5 chip connects to all 8 memory controllers, which enables each chip to have access to all of the coprocessor's memory.

believe 200 GFLOPS in achievable.

Several different HPC FPGA platforms exist. We use the Convey HC-2 (Figure 2.1). The basic idea of an FPGA implementation 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 an 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

As a way to reduce the number of x vector requests, we view registering intermediate y values superior to caching x vector values. Let us compare these 2 strategies with our example matrix.

First, row traversal, for this example assume that the last 4 vector values are stored in cache and then they are flushed from cache. In this scheme cached x values only get reused twice in the example. The first reused value is A_{72}.

Second, column traversal for every 4 rows, for this example 4 intermediate y values are registered. 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}

This method reuses x values 10 times. So, from our view we get $5\times$ more x vector reuse for the same amount of on chip memory. CPUs and GPUs have pipelines optimized for accumulating, so if they want to play this way they have to lose some pipeline efficiency.

2.11 Delta Compression

So far, all the matrix formats store indices as 32-bit values, but this seems wasteful if we already have some knowledge about the indices. Delta compression stores the distance between indices and can get better index compression than other formats like BSR.

The average number of bits to store a delta value is quite small (discussed in Chapter 5). Kourtis et al. (2008) introduces delta compression for CPUs but is vague on the details, however since this uses variable length encoding some uniquely decoded encoding scheme is required. For example unary codes. 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 area for decoding, but we will get to that later. The delta codes for the example in equation 2.1 is shown:

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

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

UNARY CODES: 0, 110, 110, 1110, 110, 10, 0, 110, 0, 0, 110, 110, 0, 110, 0, 10, 110, 10, 0, 110, 10, 110, 0, 0, 0

2.12 Value Compression

The same paper [Kourtis et al. (2008)] uses value compression, for the matrix values. Again the details are a little vague, but the idea is to take advantage of the fact values repeat. Even though this saves a lot of space for some matrices the results are again mediocre. Again, FPGAs can dedicate area for the decoder, and can potentially get better speedup results.

2.13 Benchmarking

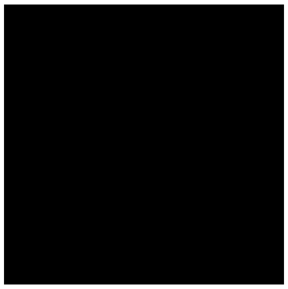
OK, now that we have a good background about SpMV, the platforms it can run on and optimizations for SpMV, we need a way to determine which implementation performs the best. This is when benchmarking comes in. However, different matrices can have vastly different SpMV performance. So a test set of matrices is used (Figure 2.2). In Figure 2.3 we show the performance of SpMV on CPUs, GPUs and FPGAs. As you can see the performance is very jumpy from matrix to matrix. Three factors effect the performance: dimension, sparsity, and values.

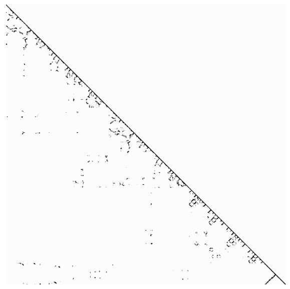
The dimension of a matrix are the height (M), the width (N) and the number of nonzeros (nnz). These metrics effect different processors differently.

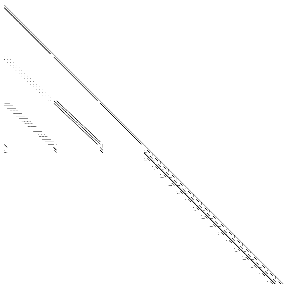
For CPUs, the values nnz and N are important. As Figure 2.3 shows when nnz is large and the matrix no longer fits in cache it takes a performance hit. It takes a second performance hit, which the figure does not show, when the width of the matrix (N) and therefore the length of the x vector grows to the point when the x vector also can not fit in cache.

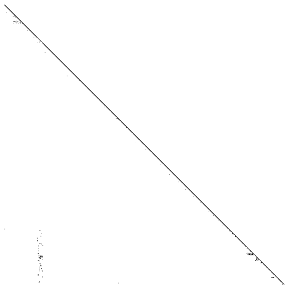
For GPUs, cache plays less of a role. However, two factors conspire against GPUs: the matrix formats they use and the amount of RAM on GPU boards. The best performing matrix formats for GPUs, like ELLPACK and Block-ELLPACK, also introduce "0" values and take up the most memory space. GPU boards currently 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.

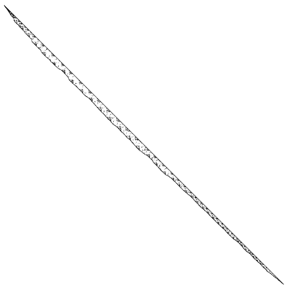
The M value also plays a role. Recall that the K40 has 30720 threads and ELLPACK uses

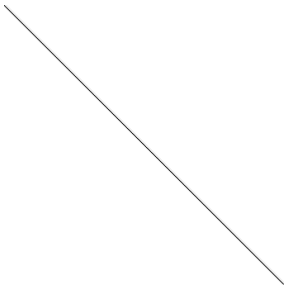

(a) dense

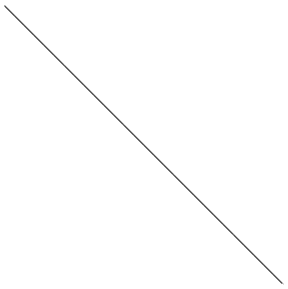

(b) pdb1HYS

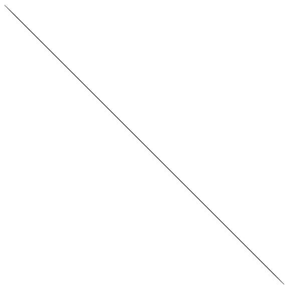

(c) consph

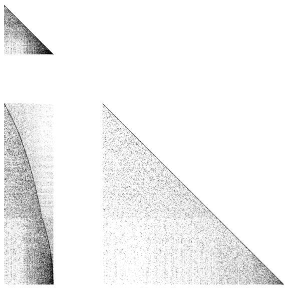

(d) pwtk

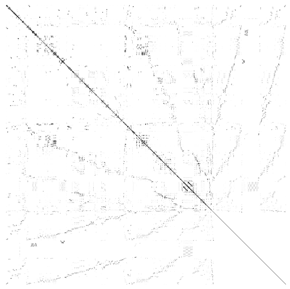

(e) shipsec1

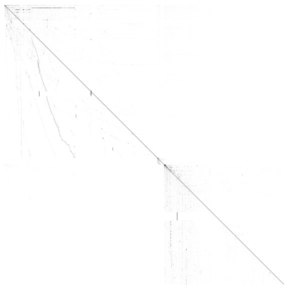

(f) mac_econ_fwd500

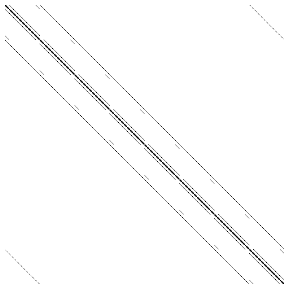

(g) cant

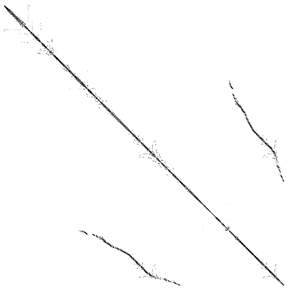

(h) mc2depi


(i) cop20k_A


(j) scircuit


(k) webbase-1M


(l) qcd5_4


(m) rma10

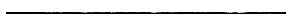

(n) rail4284

Figure 2.2: The density plots of the matrices used for testing

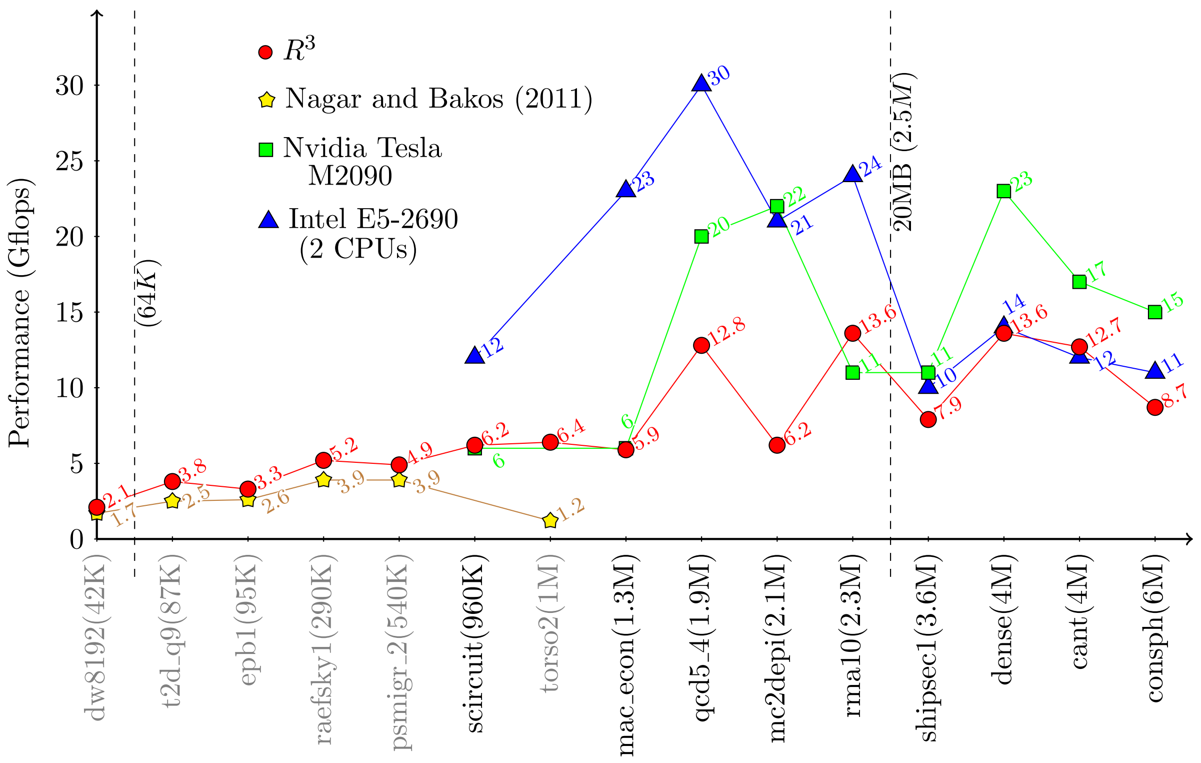


Figure 2.3: *nnz* vs Performance on each platform. The small matrices, ones around 64K or less, performed poorly on R^3, due to the overhead. CPUs experience the opposite effect. They take a performance hit once the matrix no longer fits in cache.

1 thread per row. This means the GPU is underutilized when $M < 30720$.

For FPGA implementations, like R^3, our previous SpMV implementation, nnz value plays a role. The Convey HC-2 has a long memory latency so this meant small matrices ($nnz < 64000$) would still take a couple thousand clock cycles to complete or around 0.01ms.

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. We use the Convey HC-2, however our high level design changes slightly from our R^3 design (Figure 3.1).

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 hardware amenable decompression.

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

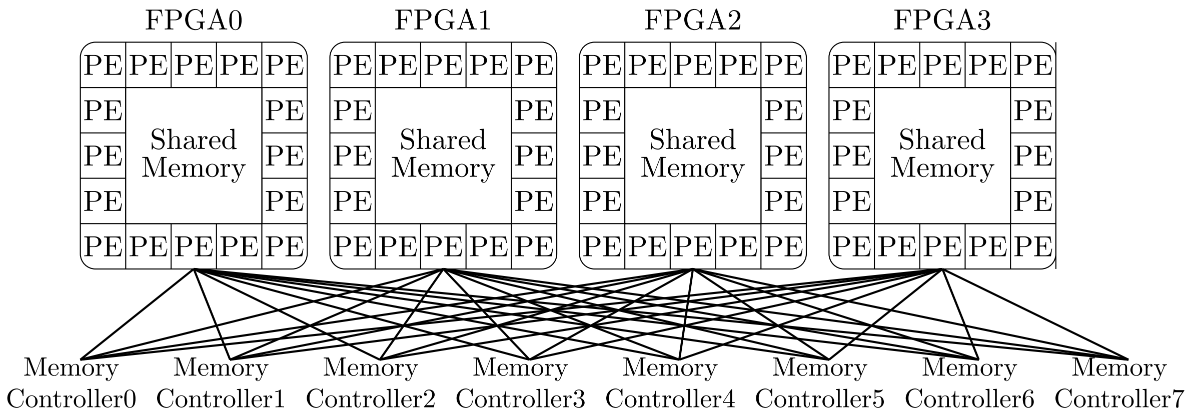


Figure 3.1: Our new implementation will have share memory for storing repeating values in the sparse matrices.

matrix element in the traversal.

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. The current view in the SpMV field does not count preprocessing of the matrix towards the SpMV runtime. This is because SpMV is usually used in 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 3.2 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 (Figure3.3). The architecture diagram also follows the same general flow of the dataflow diagram (Figure3.4).

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

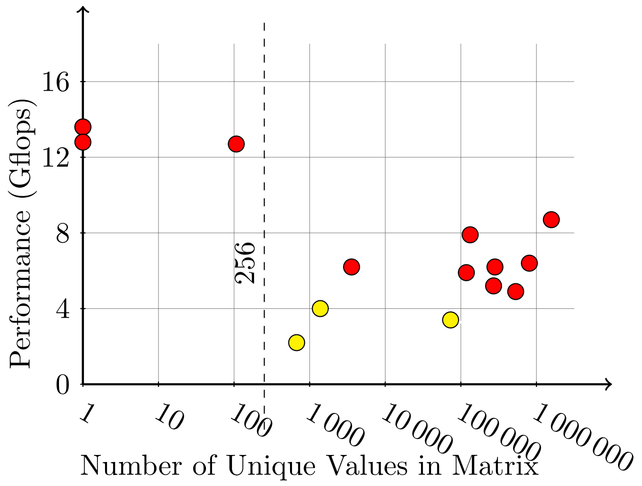


Figure 3.2: Unique values in a matrix vs the performance of R^3. Matrices with fewer than 256 unique values (only common elements exist) enables R^3 format to compress much better. The ●'s are outliers due to their size (see Figure 2.3).

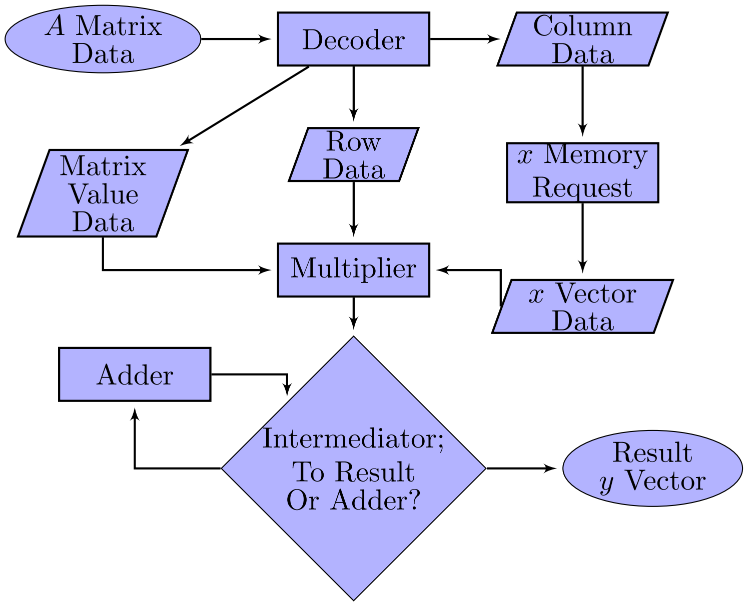


Figure 3.3: Dataflow of an SpMV processing element. The processing element needs the column data before accessing the vector data.

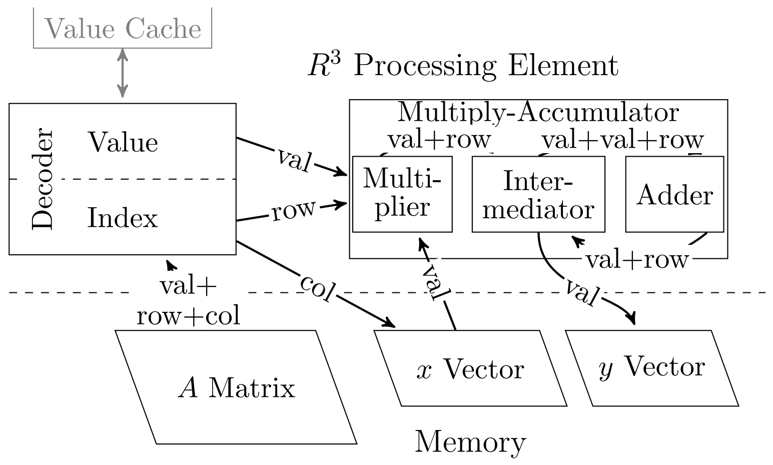


Figure 3.4: A single processing element. The arrows show the flow of data through the processing element. Although this diagram shows the memory access to each of the 3 places in memory as separate, they share one memory port. The diagram also does not show the FIFOs that help keep the pipeline full.

focuses on pillars 2 and 3 with discussing matrix traversal and compressing indices with delta compression. Chapter 6 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 7 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 complicated. To solve this, our approach works on multiple intermediate y vector values and does the additions out of order (Figure 4.1). So, $y[i]$ can still be accumulating while processing $y[i+1]$. For an example of out of order addition, when computing $1+2+3+4$ the MAC does $(1+2)+(3+4)$. This removes the data dependency of adding 1 and 2 before processing 3. CPUs and GPUs compute floating point addition in order (eg. $((1+2)+3)+4$). This means results may differ slightly, because changing the order of floating point addition can change the result [Goldberg (1991)].

In R^3 [Townsend and Zambreno (2013)] we designed a block called an Intermediator 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 RAM blocks 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

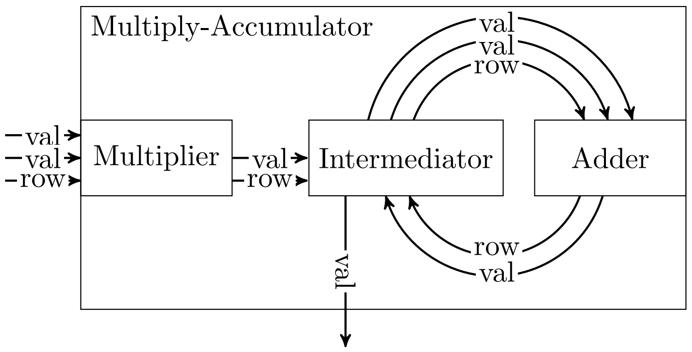


Figure 4.1: The no-stall multiply-accumulator block handles multiple intermediate values at a time. This allows multiple intermediate values in the adder pipeline.

and the MAC still works 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 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.

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 together. The dual-port RAM block (the middle block in Figure 4.2) stores intermediate values until an element in the same row appears.

For large matrices, the RAM blocks cannot store the entire intermediate y vector. Also the control logic needs to remember the state of each slot in RAM (vacant or occupied). Remembering the state of each RAM location and updating that state requires a dual-port RAM with zero clock cycle latency. This type of RAM does not exist in the FPGA fabric. In R^3, we approach this problem by using FPGA logic which limits the number of active intermediate values to 32. In our new design we will use distributed RAM with a width of 1 bit to keep track of the state of each slot (discussed later in Section 4.2).

The cells in the intermediator have four states which we call the red, yellow, green and white states, in addition to the vacant/occupied state. The memory is split in 2, an upper and lower section. Once the accumulation starts, one of these sections will be in the red active state. Once the incoming values move to the next 512 row section of the matrix this section transitions to the yellow fading state, and the new section of the memory is now in the red 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 within 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 at most 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 operation. Only inputs from the adder correspond with to the elements in the fading window and the multiplier should not output values belong to 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 no fading elements get sent to the adder and 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 pair of values to the adder. An overflow FIFO (explained in case 6) outputs a value if it has values.

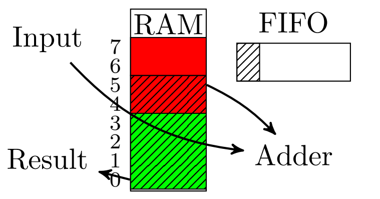
Case 2: (Figure 4.2d) Only one value arrives and the row corresponds to a vacant cell. The value goes into the vacant cell. If the "to result" window has values, it outputs a result, and if the overflow FIFO has values it outputs a set to the adder.

Case 3: (Figure 4.2a) Similar to case 2 except with an occupied cell. It retrieves the value in the Intermediator cell and goes to the adder with the input value. The state of the cell gets updated to vacant.

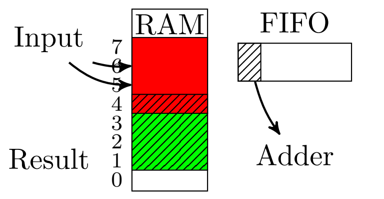
Case 4: (Figure 4.2b, 4.2i) Both values have row indexes that correspond to vacant cells in the RAM block. Both values get stored in the RAM block and both cells switch to occupied. If the overflow FIFO has values it sends one set of values to the output.

Case 5: (Figure 4.2f) One value has a row index corresponding to a vacant cell, and the other to an occupied cell. The first value goes in the vacant cell and the value in the occupied cell goes to the adder with the second value.

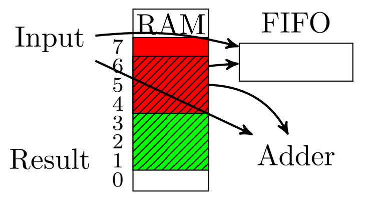
Case 6: (Figure 4.2c) Both values have row indexes that correspond to occupied cells in the



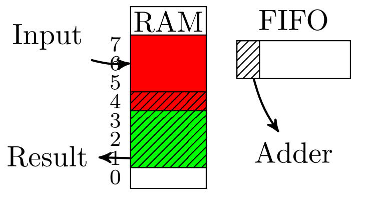
(a) First clock cycle, 1 pair of values get sent to the adder.



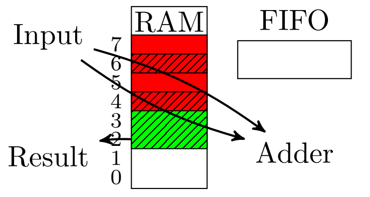
(b) Second clock cycle, 2 values gets stored in the RAM.



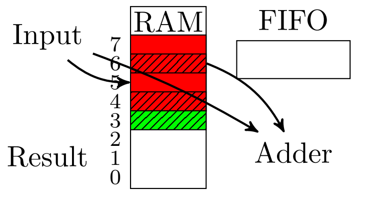
(c) Third clock cycle, the 2 inputs correspond to occupied cells in the RAM block.



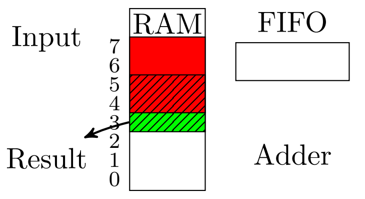
(d) Fourth clock cycle, 1 value gets stored in RAM.



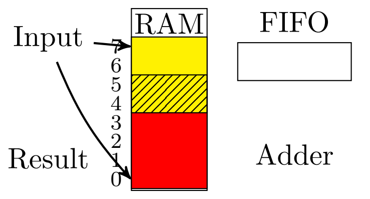
(e) Fifth clock cycle, the row indexes of the 2 inputs equal each other.



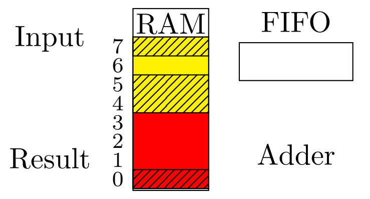
(f) Sixth clock cycle, 1 pair of values get sent to the adder, and 1 element gets stored in RAM.



(g) Seventh clock cycle, no valid inputs.



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



(i) Ninth clock cycle, the Intermediator has no inputs and nothing to send to the adder or final values to output.

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.

RAM. One input value and its corresponding Intermediator cell's value go to the output. The output can only handle one output pair at a time, so the other input value and its corresponding Intermediator cell's value go to the overflow FIFO.

Case 7: (Figure 4.2e) The values have identical row indices. In this case, the values go through the pipeline and do not touch the Intermediator cells. They simply pass through to the adder.

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.

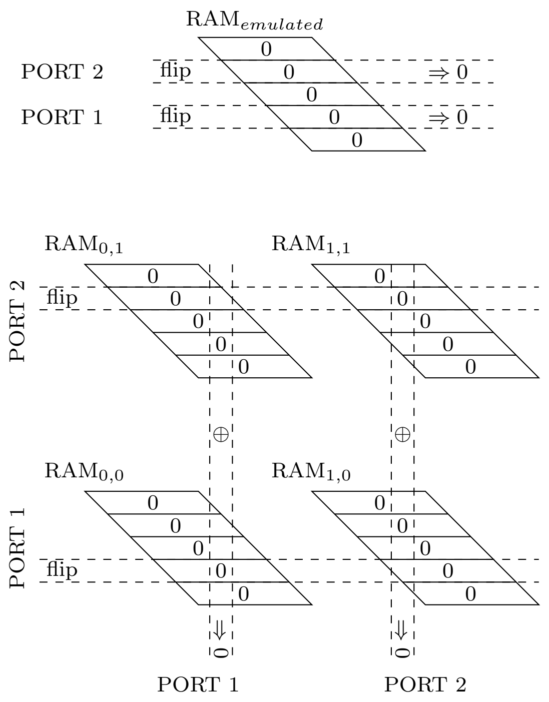
4.2 A Dual Port 1×1024 RAM with Zero Clock Cycle Latency

We have not yet addressed the issue of keeping track of the vacant/occupied state of the cells in the Intermediator. To do this we need a dual port 1-bit RAM. In order to update the value in RAM in one clock cycle it must have no read latency.

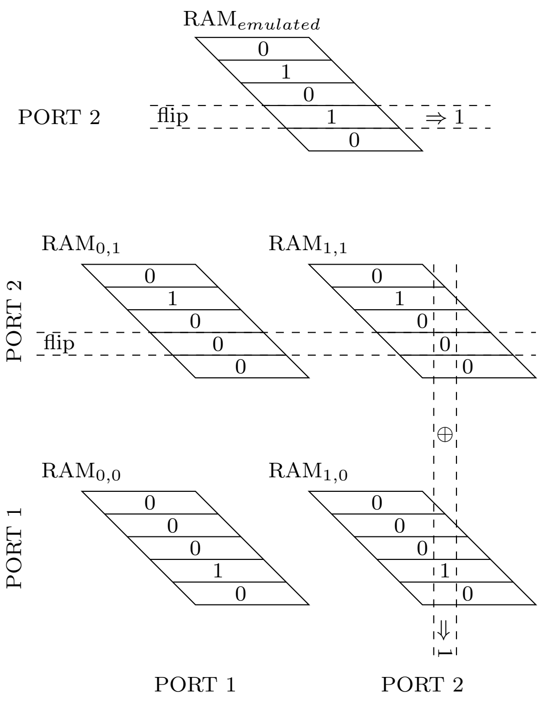
We implement a special case of the memory developed in Laforest et al. (2012) to achieve this. This requires the use of pseudo dual port distributed RAMs. Before looking at the implementation let us look at the target behavior. During an intermidiator status request the bit of the requested address will always flip. (Vacant cells become occupied and occupied cells become vacant.) This flip occurs the clock cycle after the 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.

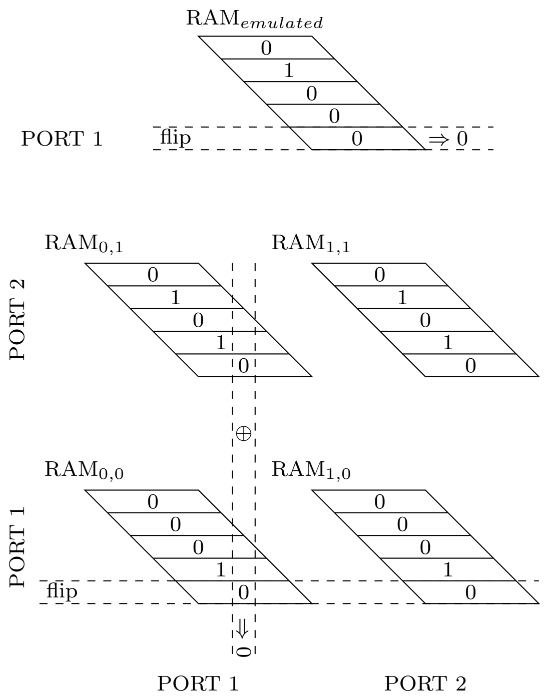
With a clever arragement of 4 pseudo dual port RAMs we can emulate one dual port RAM. To begin with, arrange the RAMs in a 2×2 grid. The write ports of the 2 RAMs in each row are connected together. The read ports of the 2 RAMs in each column are connected by an XOR gate. The address on Port 1 controls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



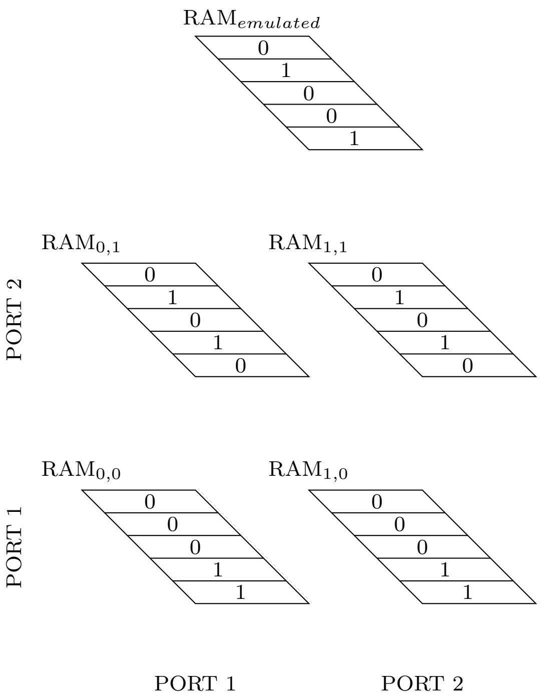
(a) On the first clock cycle, Port 1 receives a request for cell 1 and Port 2 receives a request for cell 3.



(b) On the second clock cycle, Port 2 receives a request for cell 1 and Port 1 does not receive a request.



(c) On the third clock cycle, Port 1 receives a request for cell 0 and Port 2 does not receive a request.



(d) On the fourth clock cycle, neither Port 1 or Port 2 receives a request.

Figure 4.3: This example shows 4 clock cycles of operation of a 1×5 dual port RAM, created by combining 4 pseudo dual-port RAMs.

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. In R^3, we made the mistake of combining the indices and values of sparse matrices into one compression scheme.

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

This chapter will 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, which we discussed in Section 2.10.

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

5.1 Delta Compression

Many papers use delta compression Townsend and Zambreno (2013); Kourtis et al. (2008). Delta compression stores the distance between the previous and current element. This results in smaller values that require fewer bits. The overhead of encoding the bit lengths 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 5.1 (this does not include overhead).

We also choose to use the general compression program gzip in the comparison as well. Again table 5.1 shows the compression of gzip on top of the CSR format. gzip does very well,

Table 5.1: Detailed analysis of index compression

Matrix	**COO**	**CSR**	**CSR.gz**	**Row major**[b]	**Column row-16**[b]	**Column row-256**[b]	**Column row-1024**[b]	**Experimental w/column row-16**
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

[b] Only counting delta bits.

in one case (webbase-1M) even takes less space than storing only delta bits. This is particularly surprising considering that extra overhead is needed to decode these delta bits. The reason this occurs is because large delta values can represent a short vertical jump. (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 5.2 shows this distribution. This is because column row traversal does make vertical steps.

5.2 Row Column Row (RCR) Traversal

Column row 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 for the example back in Equation 2.1 is:

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

5.3 Encoding Deltas

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

The easiest trend to see from the distribution of bits in Table 5.1 is that more than half of

Table 5.2: The distribution of the bit lengths required to store the delta length when using column row-16 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%
Unary Codes Implied Probability	50%	25%	12.5%	6.25%	3.13%	1.56%	0.78%	0.39%	0.20%	0.10%	0.05%

[a] Excludes dense matrix

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 6 bit deltas and a 7 bit delta can both be encoded as a 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 due to group size. Groups of 3 seems 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 [Saloman and Motta (2010)].

We could use the following encoding: 0 bits : 1_2, 1-3 bits : 10_2, 4-6 bits : 100_2, 7-9 bits : 1000_2 ect. In other words: a one followed by N zeros, where N is the N^{th} group of 3 delta lengths. The last column of Table 5.1 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 in fzip except the Burrows Wheeler Transform (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 [Welch (1984)], which is a little bit easier to get into hardware.

6.1 Related Work

It was noted in Kourtis et al. (2008) 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.

We analyze gzip, bzip and FPC to see how high a compression ratio over the uncompressed 8 bytes per value is achievable.

Uncompressed data would take 8 bytes per element. Any compression scheme should take less than 8 bytes per element. We looked at Burtscher and Ratanaworabhan (2009) 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

Table 6.1: Detailed value compression analysis and performance comparison

Matrix	**uncompressed**	**Unique Values**	**Unique/nnz ×8**	**256 Common**	**GZIP**	**FPC**
dense2[a]	8.00	1.00	0.00	1.00	0.01	0.50
pdb1HYS	8.00	1.10×10^6	4.08	7.99	4.15	7.99
consph	8.00	1.24×10^6	3.28	7.99	5.10	7.95
cant	8.00	1.07×10^2	0.00	1.00	0.11	0.91
pwtk	8.00	3.63×10^6	5.04	7.95	4.29	7.37
rma10[a]	8.00	1.00	0.00	1.00	0.01	0.50
qcd5_4[a]	8.00	1.00	0.00	1.00	0.01	0.50
shipsec1	8.00	8.86×10^4	0.56	6.39	2.08	3.80
mac_econ_fwd500	8.00	1.08×10^5	1.36	5.20	0.73	1.45
mc2depi	8.00	3.58×10^3	0.00	4.94	1.24	5.01
cop20k_A	8.00	9.56×10^5	5.84	7.97	5.53	7.97
scircuit	8.00	8.82×10^4	1.44	5.41	1.95	3.68
webbase-1M	8.00	5.65×10^2	0.00	1.48	0.38	1.92
average[b]	8.00	7.22×10^5	2.16	5.63	2.56	4.81

[a] Boolean matrices

[b] Excludes boolean matrices

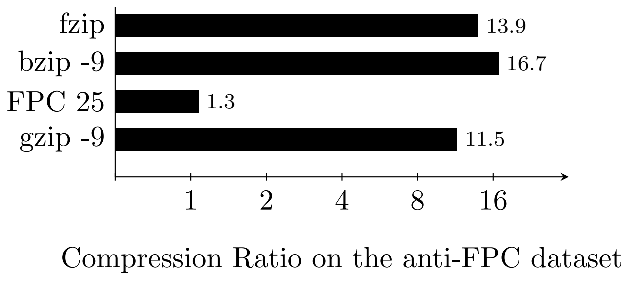


Figure 6.1: We engineering a dataset to make the performance of FPC look bad compared to other programs. Although unfair, this shows a type of pattern that FPC does not exploit and other programs do. This problem exists because FPC only uses predictors for compression.

repeats. For fun we created an "anti-FPC" dataset (Figure 6.1)

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.

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 chapter we talk about an analysis of floating point datasets (Section 6.2), our approach to floating point compression (Section 6.3) and our results (Section 6.4).

6.2 Floating-Point Value Analysis

Continuing the analysis from the beginning of this chapter, Figure 6.2 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 more than 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

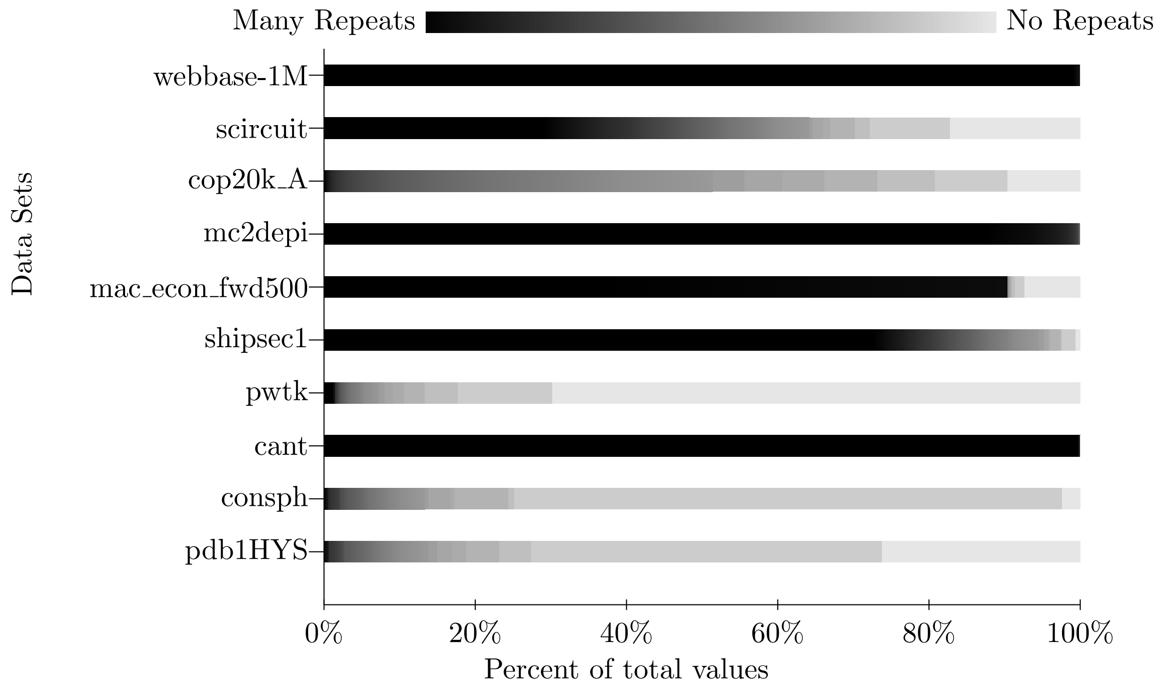


Figure 6.2: The above figure shows the distribution of repeats in each dataset. Each shade represents a different number of repeats. For instance: ■:> 512, ■:16, ■:2, ■:1(no repeats).

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 TransformBurrows and Wheeler (1994) to analyze these patterns. Figure 6.4 describes this algorithm some, however, many other sources describe this algorithm in more detailBurrows and Wheeler (1994); Saloman and Motta (2010). Figure 6.5 analyzes the number of repeats that appear after the Burrow-Wheeler Transform. As the figure shows, BWT reveals patterns in about half of the matrices.

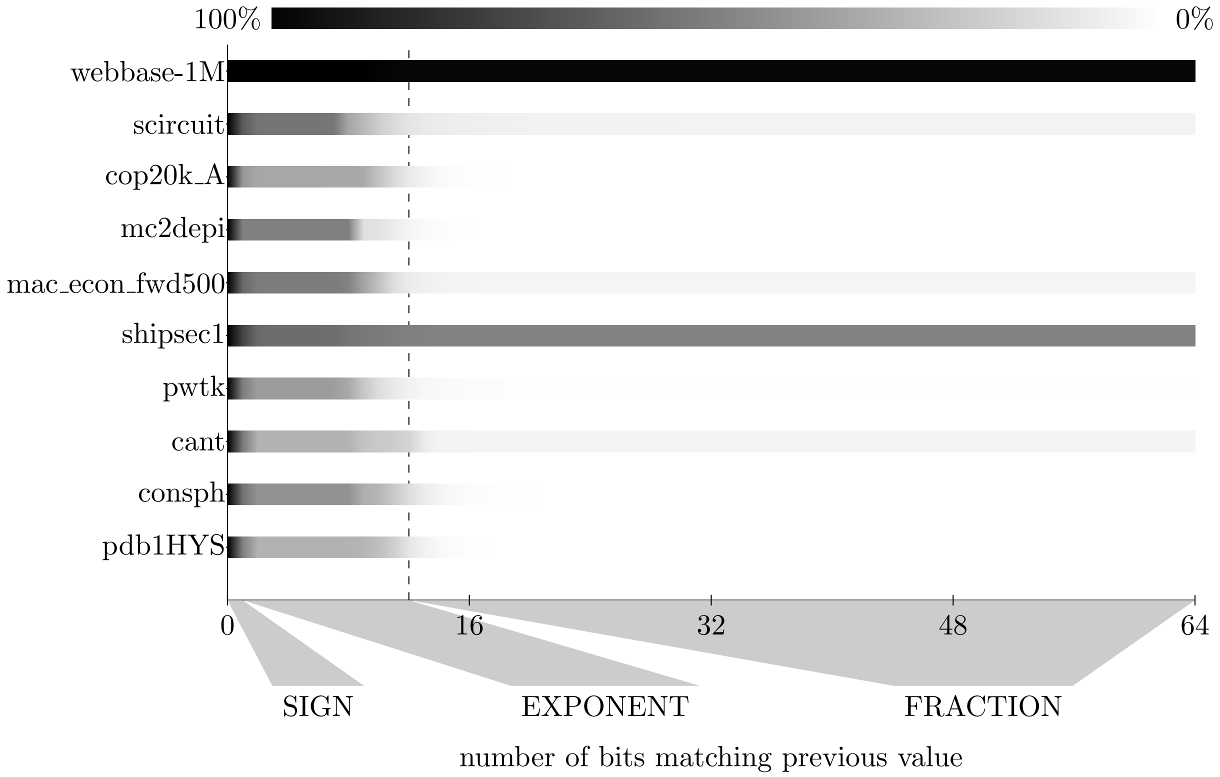


Figure 6.3: The above figure represents local prefix prediction. The figure shows the density function of 2 adjacent values sharing at least x number prefix bits. All of the data sets start at $(0, 100\%)$. The curves end at the percent of values that are identical to their previous value for that dataset.

ABCDEABCDEABC\$
\$ABCDEABCDEABC
C\$ABCDEABCDEAB
BC\$ABCDEABCDEA
ABC\$ABCDEABCDE
EABC\$ABCDEABCD
DEABC\$ABCDEABC
CDEABC\$ABCDEAB
BCDEABC\$ABCDEA
ABCDEABC\$ABCDE
EABCDEABC\$ABCD
DEABCDEABC\$ABC
CDEABCDEABC\$AB
BCDEABCDEABC\$A

(a) Step1: Generate every cyclic rotation of the original sequence (the first row).

ABCDEABCDEABC\$
ABCDEABC\$ABCDE
ABC\$ABCDEABCDE
BCDEABCDEABC\$A
BCDEABC\$ABCDEA
BC\$ABCDEABCDEA
CDEABCDEABC\$AB
CDEABC\$ABCDEAB
C\$ABCDEABCDEAB
DEABCDEABC\$ABC
DEABC\$ABCDEABC
EABCDEABC\$ABCD
EABC\$ABCDEABCD
\$ABCDEABCDEABC

(b) Step2: Sort the rotations. Then the last element in each new row creates the transformed sequence.

\$EEAAABBBCCDDC

(c) Step3: This transformed sequence often has consecutive repeats. (This example does.)

Figure 6.4: Above shows the Burrows-Wheeler Transform and subsequent compression. Steps 1 and 2 show the brute force calculation of BWT.

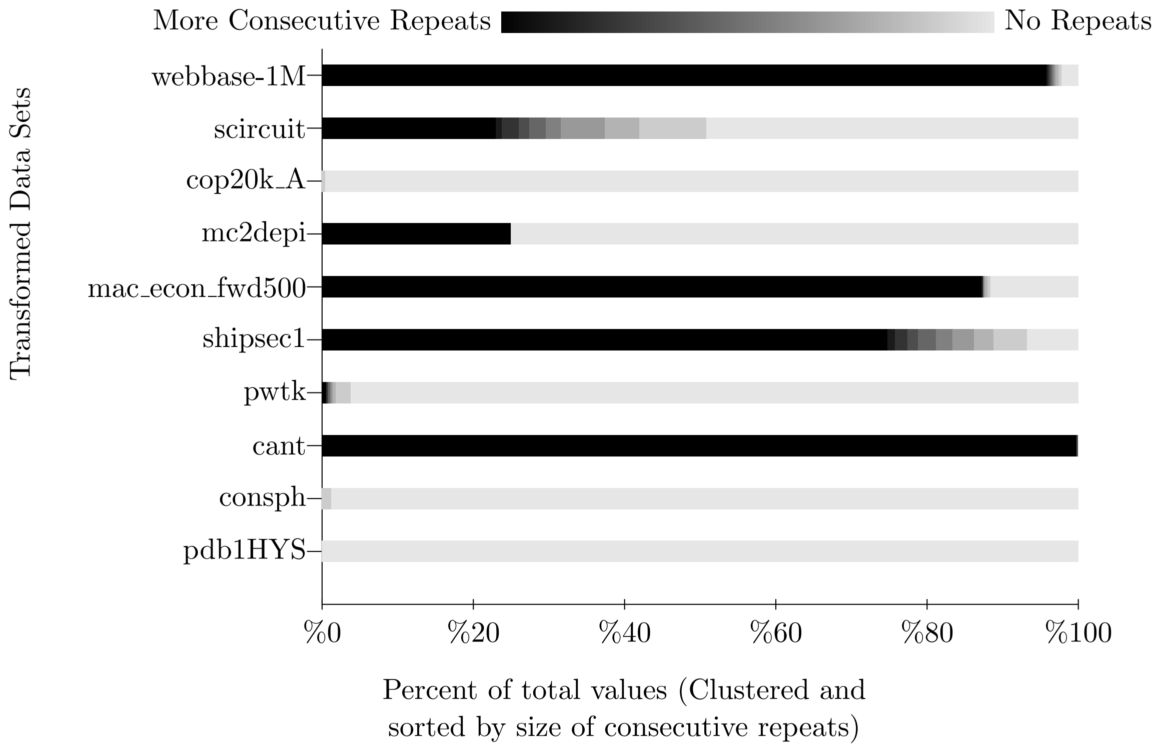


Figure 6.5: Pattern analysis using the Burrows-Wheeler Transform. Each shade represents the number of consecutive repeats in a repeating sequence. ■ represents sequences longer than 9. ▨ represents sequences of length 5. □ represents sequences equal to 1 (non-repeating).

6.3 Our Approach

Since BWT does not provide great compression and is not hardware amenable, we will configure fzip to only use prefix and repeat compression.

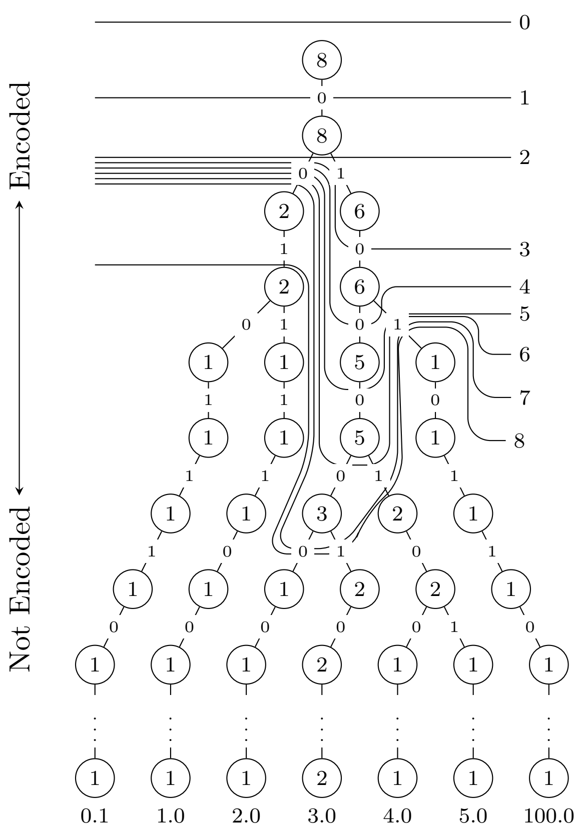
6.3.1 Prefix Compression

fzip uses arithmetic codes to encode common prefixes. To begin with, fzip creates a large tree to represent all the values in the array. Figure 6.6a 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).

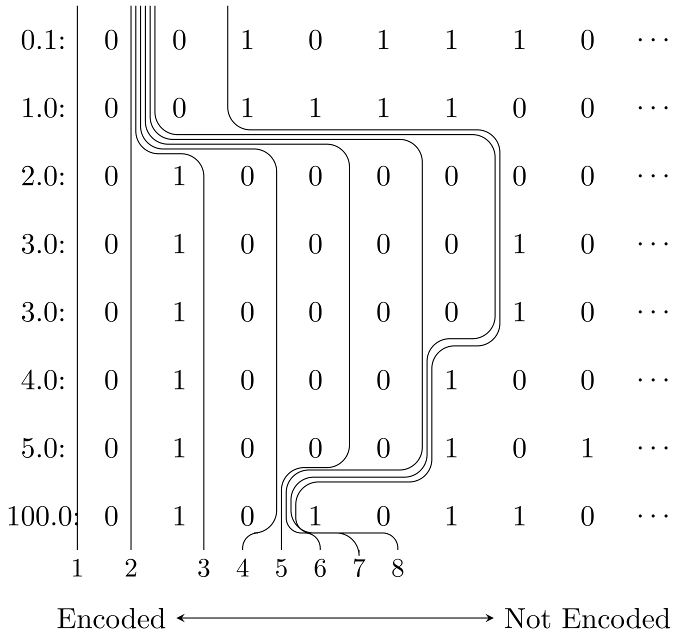
Several properties appear. First, the sum of all the weights of the nodes in any level equals nnz, where nnz is the total number of values. Moreover, the weight of any set of nodes that partitions the root node from the 65^{th} level (and does not contain more nodes than necessary to create the partition)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

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.



(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 6.6: 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.

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 node with the largest weight that is a child of the partition. This repeats until a predetermined number of edges become cut by the partition. Using a list of prefix, prefix code pairs we can represent the encoding scheme of the first 8 partitions of the example in Figure 6.6:

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 $\frac{nnz}{k}$, where nnz equals to number of values in the data set. If you choose A to maximize weight(A) (a greedy algorithm) then weight(A) $>$ average weight of children to the partition $= \frac{nnz}{k}$. 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.

6.3.2 Repeated Value Extension

Prefix compression does not compress all of the repeated values. So, fzip extends prefix compression to specifically include all repeated values. Again explaining why repeated values compress well: All of the datasets have less than 6 million values. An index of 23 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 $23 + 23 + 64 < 64 + 64$ (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.

6.4 Discussion

We use fzip's results to get an idea of performance (Figure 6.7). Currently, we do not have the hardware amenable version of fzip ready. As Chapter 9 describes, we plan to have a hardware decoder for fzip by September.

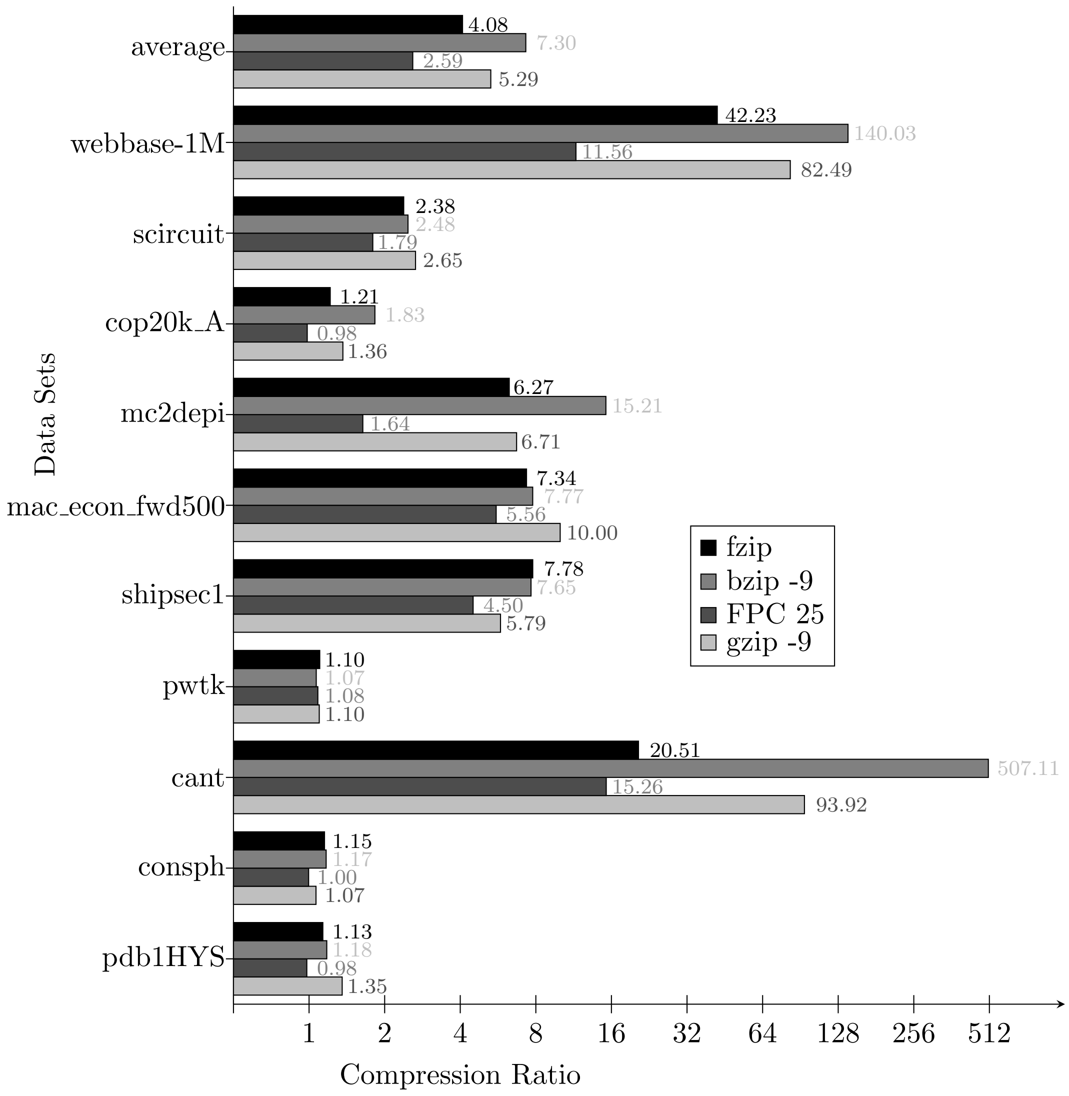


Figure 6.7: The comparison of different compression schemes shows fzip performs competitively.

CHAPTER 7. MULTI-PORT RAM

The repeated values array requires a large amount of space. To efficiently store this array on the FPGA we created 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.

7.1 Related Work

FPGAs have RAM blocks for designs that require large amounts of memory space. Four strategies exist for creating multi-port memory with RAM blocks: *multi-pumping*, *replication*, *Live Value Table*, and *banking*.

Multi-pumping, seen in Manjikian (2003); Canis et al. (2013); Yantir et al. (2013), cannot support our desired clock frequency. *Replication*, seen in Fort et al. (2006); Moussali et al. (2007); Yiannacouras et al. (2006), and *Live Value Table*, seen in LaForest and Steffan (2010); Anjam et al. (2010); Abdelhadi and Lemieux (2014), store excessive redundant information information in RAM blocks. This leaves *Banking*, seen in Moscola et al. (2010); Saghir and Naous (2007); Saghir et al. (2006), which can scale to 16 or more ports.

A straight forward way to create this memory would use full-connected interconnect networks (Figure 7.1). 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.

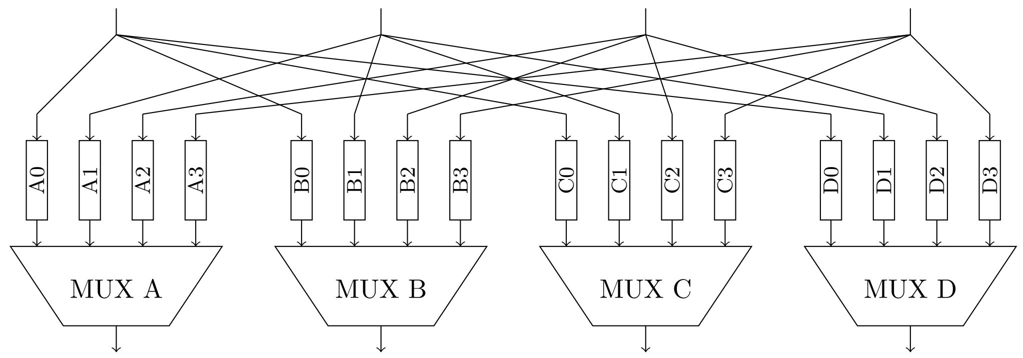


Figure 7.1: Fully-connected interconnect network

7.2 Omega Multi-port Memory

The Omega multi-port memory (Figure 7.2) 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.

The Omega memory has several subcomponents: first, the memory banks for storing the data, second, Omega networks for routing between the ports and banks, third, linked list FIFOs to buffer requests to banks, fourth, reorder queues to reorder read responses.

7.2.1 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

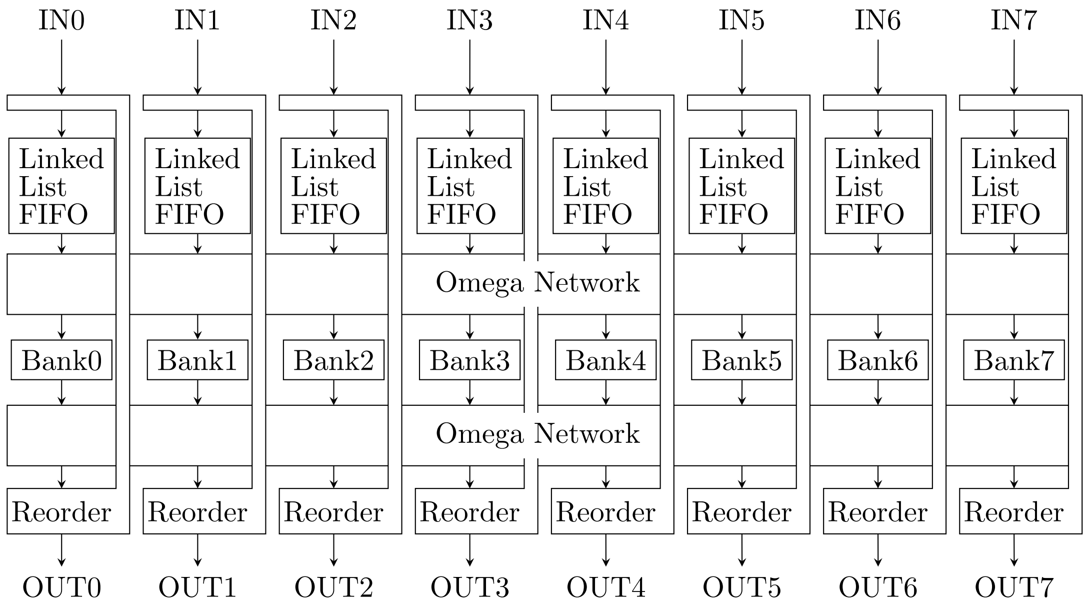


Figure 7.2: In the Omega multi-port memory all the buffering occurs in the linked list FIFOs. The use of multi-stage interconnect networks, in this case Omega networks, helps reduce the area of the design.

space that our design uses decreases the chance these specific types of bottlenecks occur.

7.2.2 Omega Network

An Omega network consists of columns of Banyan switches Wu and Feng (1980); Lawrie (1975). 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 Figure 7.3 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 Figure 7.3 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 Figure 7.3, 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

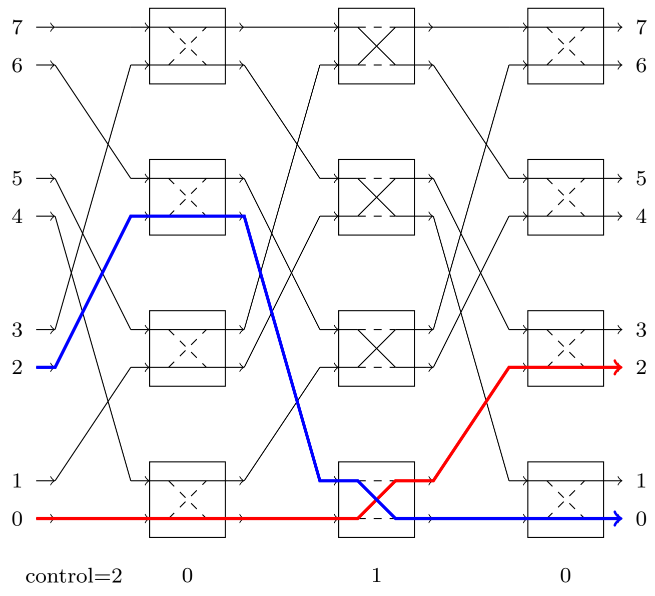


Figure 7.3: An 8-by-8 Omega network. We turn columns on or off to rotate between different routing configurations.

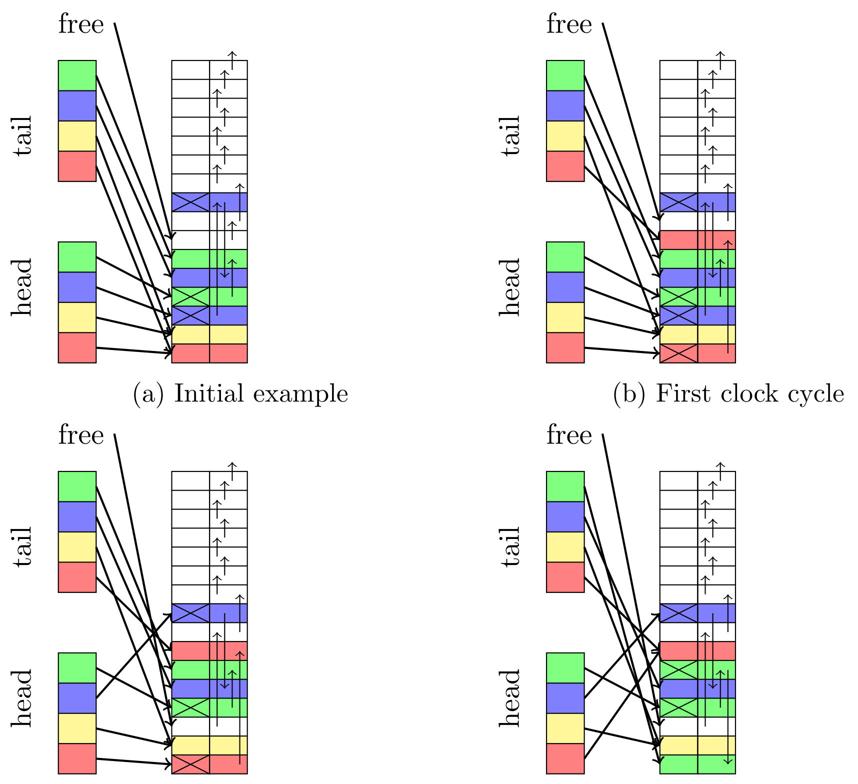
bits for both the receiving and sending Omega networks.

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.

7.2.3 Linked List FIFO

The partnering hardware structure, the linked list FIFO (Figure 7.4), 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. Other variants of hardware linked list FIFOs exist [Bell et al. (2008); Nikologiannis et al. (2004)].

Due to the linking pointers, the size of the RAM now needs $O(N \log N)$ space to store N elements. However $\log N$ 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 Figure 7.4, which uses a 16 deep RAM and 4 FIFOs.



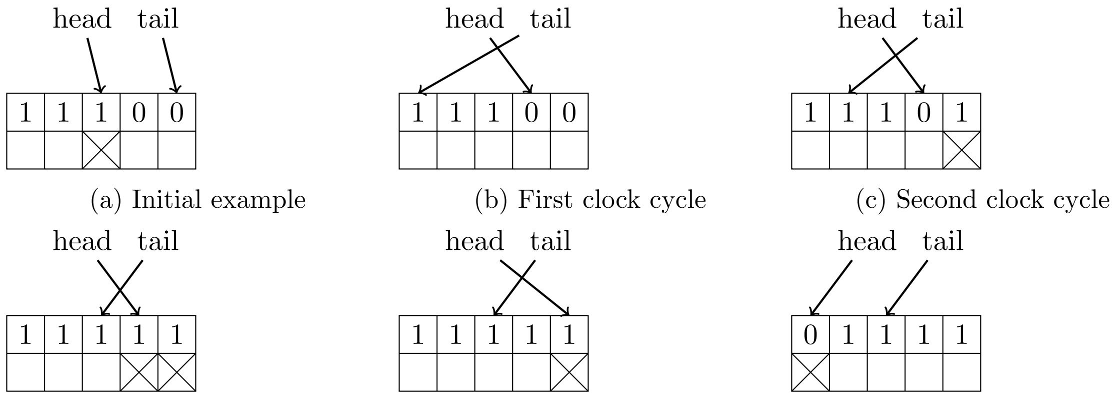
(a) Initial example

(b) First clock cycle

(c) Second clock cycle

(d) Third clock cycle

Figure 7.4: 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 7.5: Reorder queue example.

In the initial state (Figure 7.4a), 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 (Figure 7.4b), the linked list FIFO receives a push containing a red 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 (Figure 7.4c), 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 (Figure 7.4d), 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.

7.2.4 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 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 7.5 shows an example with 5 clock cycles of operation. In the initial state (Figure 7.5a), the reorder queue has one present message and one in flight message.

On the first clock cycle (Figure 7.5b), 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 (Figure 7.5c), 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 (Figure 7.5d), a message arrives at the head of the reorder queue.

On the fourth clock cycle (Figure 7.5e), 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 7.5f), 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.

7.3 Evaluation Methodology

We limit linked list FIFOs and reorder queues to a depth of 64. We limit the depth of the FIFOs in the fully-connected interconnect network to 32 since the number of FIFOs in it grows by $O(N^2)$.

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 sequential, random, congested, or segregated memory access patterns.

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

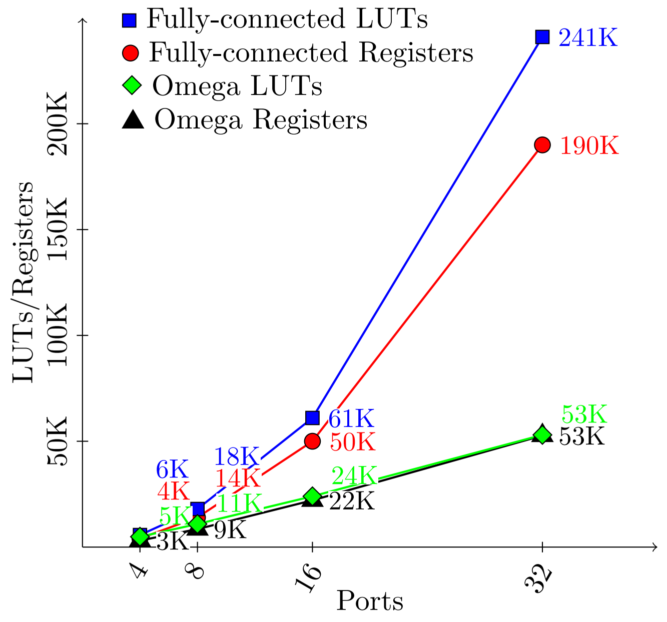


Figure 7.6: The effect of varying the number of ports on FPGA resource utilization (area). The Fully-connected memory grows by approximately N^2 and the Omega memory grows almost linearly.

latency by measuring the number of clock cycles between the last read request and the last read response.

7.4 Results and Analysis

We present the results of the Omega memory in Table 7.1 and include results from a Fully-connected memory as a comparison.

7.4.1 Varying the number of ports

In terms of area, Figure 7.6 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.

In terms of performance, Figure 7.7 shows that increasing the number of ports decreases

Table 7.1: Analysis of the Omega multi-port memory design

Ports Memory Space		4 16KB	8 32KB	16 64KB	32 128KB
	Fully-connected Multi-port Memory				
Resource Utilization Virtex 7 V2000T[2]	Registers	4K	14K	50K	190K
	LUTs	5.7K	18K	61K	241K
	BlockRAM	4	8	16	32
	Clock frequency	345Mhz	313Mhz	256Mhz	273Mhz
Sequential	Throughput	100%	100%	100%	100%
	Latency [1]	16	20	36	64
Random	Throughput	97%	93%	88%	72%
	Latency [1]	66	65	85	97
Congested	Throughput	25%	13%	6%	3%
	Latency [1]	105	230	490	1034
Segregated	Throughput	100%	100%	100%	100%
	Latency [1]	16	24	34	63
	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

[1] This measures the number of clock cycles between the end of the benchmark and when the last response of the last request gets received. In the worst case scenario several FIFOs queue data that has to wait for access to the same bank.

[2] This particular chip has 2.4M registers, 1.2M LUTs and 1.3K RAM blocks.

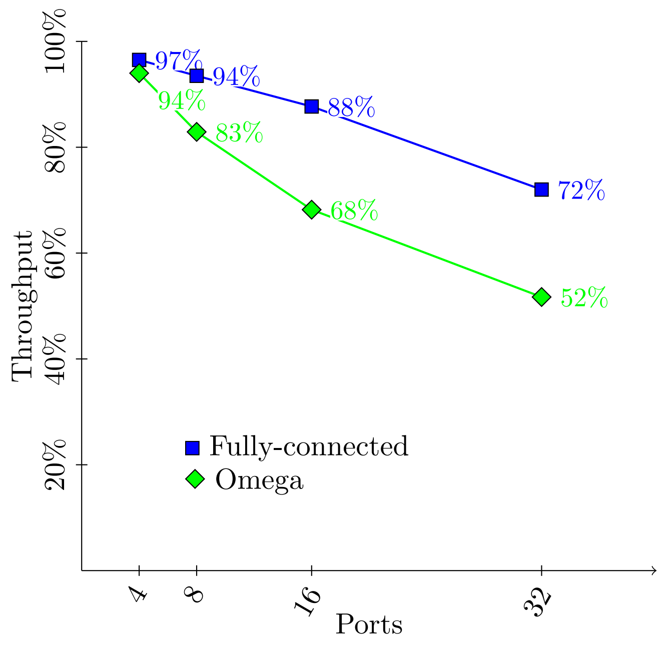


Figure 7.7: The effect of varying the number of ports on throughput of the random memory access benchmark on the small resource memories.

throughput, and Table 7.1 shows increasing the number of ports increases latency. As expected, throughput decreases a little faster for the Omega memory. The latency grows almost linearly with the number of ports, because of the round robin contention resolution scheme. On average it takes $\frac{N}{2}$ clock cycles to start processing the first memory request.

7.4.2 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 7.8 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.

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

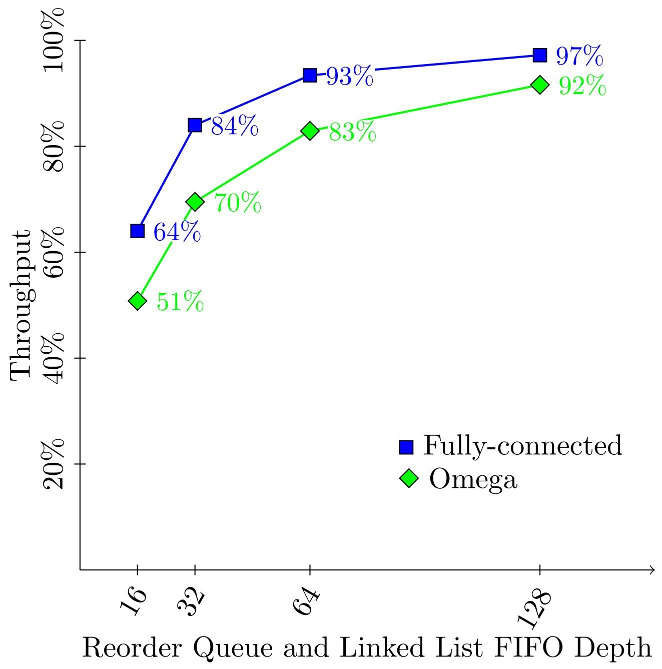


Figure 7.8: The effect of varying the depth of the linked list FIFOs and reorder queues on throughput of the random memory access benchmark (using 8-port memories).

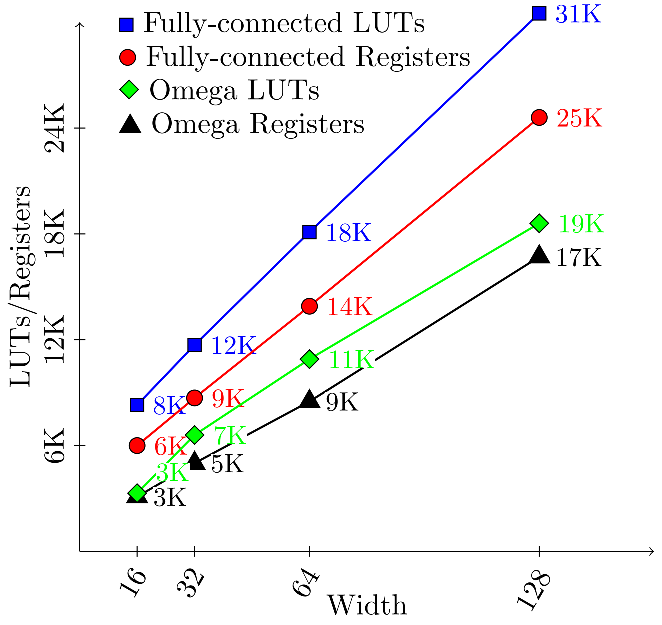


Figure 7.9: The effect of varying the bit-width of the memory on FPGA resource utilization.

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. But, if buffers consist entirely of distributed RAMs (LUT resources), FPGA utilization increases linearly with the buffer depth.

7.4.3 Varying the data bit width

Data width only effects resource utilization. As Figure 7.9 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 16 ports is 35 GB/s.

CHAPTER 8. RESULTS

R^3, our previous work, achieved up to 12 GFLOPS and an average performance of 8 GFLOPS on the non-pattern matrices used by Bell and Garland (2008). We were able to double this performance to an average of 16 GFLOPS.

CHAPTER 9. CONCLUSIONS

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

9.1 Related Work

More information about the work presented in this report is in the following papers:

- "A Scalable Unsegmented Multi-port Memory for FPGA-based Systems" (in submission) (paper available on request)
- "A Multi-Phase Approach to Floating-Point Compression" Townsend et al. (2015)
- "Reduce, Reuse, Recycle (R^3): A Design Methodology for Sparse Matrix Vector Multiplication on Reconfigurable Platforms" Townsend and Zambreno (2013)

BIBLIOGRAPHY

Abdelhadi, A. M. S. and Lemieux, G. G. F. (2014). Modular multi-ported SRAM-based memories. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programable Gate Arrays (FPGA)*, pages 35–44.

Anjam, F., Wong, S., and Nadeem, F. (2010). A multiported register file with register renaming for configurable softcore VLIW processors. In *Proceedings of the International Conference on Field-Programable Technology (FPT)*, pages 403–408.

Bell, N. and Garland, M. (2008). Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-004, Nvidia.

Bell, S., Edwards, B., Amann, J., Conlin, R., Joyce, K., Leung, V., MacKay, J., Reif, M., Bao, L., Brown, J., Mattina, M., Miao, C., Ramey, C., Wentzlaff, D., Anderson, W., Berger, E., Fairbanks, N., Khan, D., Montenegro, F., Stickney, J., and Zook, J. (2008). TILE64$^{\text{TM}}$ processor: A 64-core SoC with mesh interconnect. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 88–598.

Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossess data compression algorithm. Technical Report 124, Systems Research Center.

Burtscher, M. and Ratanaworabhan, P. (2009). Fpc: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers (TC)*, 58(1):18–31.

Canis, A., Anderson, J. H., and Brown, S. D. (2013). Multi-pumping for resource reduction in FPGA high-level synthesis. In *Proceedings of the IEEE Design, Automation & Test in Europe (DATE)*, pages 194–197.

Cappello, J. D. and Strenski, D. (2013). A practical measure of FPGA floating point acceleration for high performance computing. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 160–167.

Choi, J. W., Singh, A., and Vuduc, R. W. (2010). Model-driven autotuning of sparse matrix-vector muliply on GPUs. pages 115–126.

Davis, J. D. and Chung, E. S. (2012). Spmv: A memory-bound application on the gpu stuck between a rock and a hard place. Technical Report MSR-TR-2012-95, Microsoft.

Fort, B., Capalija, D., Vranesic, Z. G., and Brown, S. D. (2006). A multithreaded soft processor for SoPC area reduction. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 131–142.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48.

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

Laforest, C. E., Liu, M. G., Rapati, E. R., and Steffan, G. J. (2012). Multi-ported memories for FPGAs via XOR. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programable Gate Arrays (FPGA)*, pages 209–218.

LaForest, C. E. and Steffan, J. G. (2010). Efficient multi-ported memories for FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programable Gate Arrays (FPGA)*, pages 41–50.

Lawrie, D. H. (1975). Access and alignment of data in an array processor. *IEEE Transactions on Computers (TC)*, 24(12):1145–1155.

Manjikian, N. (2003). Design issues for prototype implementation of a pipelined superscalar processor in programmable logic. In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 155–158.

Moscola, J., Cytron, R. K., and Cho, Y. H. (2010). Hardware-accelerated RNA secondary-structure alignment. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3(3):1–44.

Moussali, R., Ghanem, N., and Saghir, M. A. R. (2007). Supporting multithreading in configurable soft processor cores. In *Proceedings of the ACM International Conference on Compilers, Architecture, and Synthesis of Embedded Systems (CASES)*, pages 155–159.

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

Nikologiannis, A., Papaefstathiou, I., Kornaros, G., and Kachris, C. (2004). An FPGA-based queue management system for high speed networking devices. *Microprocessors and Microsystems (MICPRO)*, 28(5):223–236.

Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford.

Saghir, M. A. R., El-Majzoub, M., and Akl, P. (2006). Datapath and ISA customization for soft VLIW processors. In *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–10.

Saghir, M. A. R. and Naous, R. (2007). A configurable multi-ported register file architecture for soft processor cores. In *Proceedings of the ACM International Workshop on Applied Reconfigurable Computing (ARC)*, pages 14–25.

Saloman, D. and Motta, G. (2010). *Handbook of Data Compression.* Springer, London, 5 edition.

Townsend, K. and Zambreno, J. (2013). Reduce, reuse, recycle (R^3): a design methodology for sparse matrix vector multiplication on reconfigurable platforms. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP).*

Townsend, K. R., Jones, P., and Zambreno, J. (2014). A high performance systolic architecture for k-NN classification. In *Proceedings of the IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 201–204.

Townsend, K. R., Sun, S., Johnson, T., Attia, O. G., Jones, P. H., and Zambreno, J. (2015). k-NN text classification using an FPGA-based sparse matrix vector multiplication accelerator.

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

Welch, T. A. (1984). A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19.

Wu, C. and Feng, T. (1980). On a class of multistage interconnection networks. *IEEE Transactions on Computers (TC)*, 29(8):694–702.

Yantir, H. E., Bayar, S., and Yurdakul, A. (2013). In *Proceedings of the IEEE Euromicro Conference on Digital System Design (DSD)*, pages 185–192.

Yiannacouras, P., Steffan, J. G., and Rose, J. (2006). Application-specific customization of soft processor microarchitecture. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programable Gate Arrays (FPGA)*, pages 201–210.