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

Zhao Zhang

Iowa State University

Ames, Iowa

2016

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

There are hundreds of papers on accelerating sparse matrix vector multiplication (SpMV), however, only a handful target FPGAs. Many claim that FPGAs inherently perform inferiorly to CPUs and GPUs. FPGAs do perform inferiorly for some applications like matrix-matrix multiplication and matrix-vector multiplication. CPUs and GPUs have too much memory bandwidth and too much floating point computation power for FPGAs to compete. However, the low computations to memory operations ratio and irregular memory access of SpMV trips up both CPUs and GPUs. We see this as a leveling of the playing field for FPGAs.

Our implementation focuses on three pillars: matrix traversal, multiply-accumulator design, and matrix compression. First, Most SpMV impementations traverse the matrix in row-major order, but we mix column and row traversal. Second, To accomadate the new traversal the multiply accumulator stores many intermediate y values. Third, we compress the matrix to increase the transfer rate of the matrix from RAM to the FPGA. Together these pillars enable our SpMV implementation to perform competetively with CPUs and GPUs.

CHAPTER 1. INTRODUCTION

This dissertation outlines a method to achieve high performance sparse matrix vector multiplication (SpMV) on the Convey HC-2ex. Although we target one specfic platform this work should port well to other FPGA platforms. In creating our solution, we developed several technologies that reach into adjacent domains, including: a new matrix traversal, a new multiply-accumulator, a new sparse matrix compression algorithm, a new floating point compressing algorithm, and a multi-port memory core.

SpMV is used in a variety of applications including information retrieval [**?**)], text classification [**?**)], and image processing [**?**)]. Often, the SpMV operations are iterative or repetitive and require a large amount of computation. For example, the PageRank algorithm uses iterative SpMV for eigenvector estimation.

For the most part, modern CPUs compute SpMV well. On CPUs and most other platforms SpMV is a memory bound application. So the performance on CPUs will depend on the CPUs memory bandwidth or the amount of cache on the CPU. FPGAs have no where near the amount of memory bandwidth that current CPU and GPU machines have. The machine we use (the Convey HC2-ex) has only 19GB/s memory bandwidth per FPGA, whereas current CPUS have 100 GB/s and current GPUs have 290 GB/s of memory bandwidth. However, this is a little bit of a oversimplification of the problem.

There is a small niche where FPGAs can excel, even with this handicap. Application that use repetitive SpMV operations on large matrices have a chance of performing the best on FPGAs. When the matrix and vector sizes become large, around 10 million values, CPU performance drastically decreases. When this happens the CPU experiences a lot of x vector cache misses. To address this issue many turn to GPUs.

However, GPUs have an interesting characteristic. In order to achieve good performance,

GPUs expand the storage size of the matrix. FPGA implementations generally 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.

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

In Chapter **??**, we describe the previous approaches to SpMV on CPUs, GPUs and FPGAs. In Chapter **??**, we outline our approach for achieving the best possible SpMV performance on FPGAs. In Chapters **??**, **??**, **??**, **??**, and **??**, we discuss our optimizations for FPGAs. In Chapter **??**, we present our high level design and results. In Chapter **??**, 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. Consider the example SpMV operation in Equation **??**. As Equation **??** shows, matrix vector multiplication is a series of dot products.

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

$$= \begin{bmatrix} A_{11}x_1 + A_{14}x_4 + A_{17}x_7 \\ A_{25}x_5 + A_{28}x_8 \\ A_{32}x_3 + A_{33}x_3 + A_{36}x_6 + A_{37}x_7 \\ A_{41}x_1 + A_{45}x_5 \\ A_{53}x_3 + A_{54}x_4 + A_{57}x_7 + A_{58}x_8 \\ A_{62}x_2 + A_{65}x_5 \\ A_{72}x_2 + A_{73}x_3 + A_{76}x_6 + A_{78}x_8 \\ A_{83}x_3 + A_{84}x_4 + A_{85}x_5 + A_{86}x_6 \end{bmatrix} \tag{2.2}$$

Sparse matrices differ from dense matrices in that they contain mostly (usually more than 99%) zeros. This makes the number of non-zeros nnz an important measurement of sparse matrices. In most datasets nnz grows by $O(M)$ instead of $O(MN)$.

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. Sparsity of matrices is usually measured in elements per row rather than a percent. It makes more sense to say the average user has 300 facebook friends that to say the average facebook user is friends with 0.00003% of facebook users. And, the percent sparsity of the matrix will keep growing but the number of non-zero elements per row will stay relatively constant.

The reader may skip directly to Chapter **??**, which discusses our approach to SpMV on FPGAs, at any point in this chapter. The rest of the chapter discusses implementations of SpMV and their computation platforms starting with COO format in Section **??**. Then, Section **??** discusses CPUs, followed by optimizations relevant to CPUs in Sections **??**, **??**, and **??**. Then Section **??** discusses GPUs, followed by optimizations relevant to GPUs in Sections **??**, and **??**. Then Section **??** discusses FPGAs, followed by optimizations relevant of FPGAs in Sections **??**, and **??**. Lastly, Section **??** discusses benchmarking the performance of SpMV.

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 facebook matrix would have to store a quintillion (10^{18}) values if stored in dense format.) This alternative, coordinate format (COO), stores 3 arrays: a row index array, a column index array, and a value array. By popular convention, row and column indices are 4 byte (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

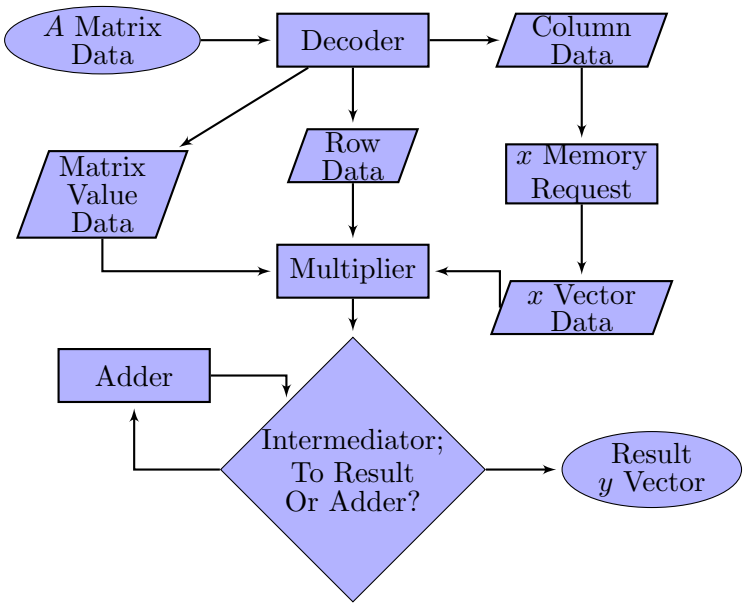


Figure 2.1: The dataflow is similar for all SpMV implementations. Often the matrix is not stored in COO format and needs be decoded into each matrix's column, row and floating point value. As the dataflow shows, the processor needs the column data before accessing the x vector data.

starts at the left most element of the first row (A_{11}), and 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}). We discuss alternate traversals later in this chapter.

Calculating SpMV with this matrix format is straight forward and much faster than if the whole dense 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 $2 \times nnz$ to report performance, which we use to report our performance. The difference is 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

Computing SpMV on any platform follows the dataflow outlined in Figure **??**. It only takes a few lines to write an SpMV function in C:

```
void spmv(double* y, double* x, int* row, int* column, double* value, int nnz,
```

```
        int height){
    // Zero the y vector.
    for(int i = 0; i < height; ++i){
        y[i] = 0;
    }
    // Compute SpMV.
    for(int i = 0; i < nnz; ++i){
        y[row[i]] = y[row[i]] + value[i] * x[column[i]];
    }
}
```

The sparsity of the matrix causes CPUs to perform below their potential. A recent Intel publication using 2 Xeon E5-2699 v3 processors show an average performance of 1 TFLOP for matrix matrix multiplication but Intel publishes an average performance of 27 GFLOPS for SpMV.

To understand this, look at Equation **??** again and count the number of times each value is accessed (appears in Equation **??**). 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) grows slowly for larger matrices. This means the computation operations to memory operations ratio is low. Compare this to matrix-matrix multiplication where the ratio is high and each CPU can perform at 500 GFLOPs, almost the limit of the CPU.

The effect of this small ratio affects the CPU less when everything can fit in cache. In these cases performance is bounded by the bandwidth of the L3 (top level) cache.

2.2.1 Compressed Sparse Row Format (CSR)

The simplest optimization over COO is compressed sparse row (CSR). This optimization compresses the row indices. The column and value arrays are the same as they were in COO, but a compressed row array replaces the row array. In COO, 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 **??** 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}

2.2.2 Block Sparse Row Format (BSR)

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

The block sparse row (BSR) storage format is one such block storage scheme. It stores the row and column indices of the top left of the block and stores the values of the block in row major form. This matrix format 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 a pessimistic view of hybrid formats, because this results in performing SpMV on 2 matrices, both of which are sparser than the original. Other papers try to minimize this negative effect by minimizing the size of the second matrix [**??**)].

The block sparse row format for the example in Equation **??** 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 of storing explicit zero values [**?**)].

2.2.3 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 subblocks individually before proceeding to the next block [**?**)]. The dimensions of the blocks are set so that the relevant subsections in the x and y vector fit in cache. This method has similarities to our row column row (RCR) traversal, introduced later in Chapter **??**. Using the example in Equation **??**, one method of cache blocking would be to spit the matrix into 4 smaller matrices:
Submatrix 1:

ROW: 0, 0, 2, 2, 3

COLUMN: 0, 3, 1, 2, 0

VALUE: A_{11}, A_{14}, A_{32}, A_{33}, A_{41}

Submatrix 2:

ROW: 0, 1, 1, 2, 2, 3

COLUMN: 6, 4, 7, 5, 6, 4

VALUE: A_{17}, A_{25}, A_{28}, A_{36}, A_{37}, A_{45}

Submatrix 3:

ROW: 4, 4, 5, 6, 6, 7, 7

COLUMN: 2, 3, 1, 1, 2, 2, 3

VALUE: A_{53}, A_{54}, A_{62}, A_{72}, A_{73}, A_{83}, A_{84}

Submatrix 4:

ROW: 4, 4, 5, 6, 6, 7, 7

COLUMN: 6, 7, 4, 5, 7, 4, 5

VALUE: A_{57}, A_{58}, A_{65}, A_{76}, A_{78}, A_{85}, A_{86}

2.3 GPU

Before discussing storage formats specific to GPUs, it is important to understand GPUs compute differently than CPUs. To show this let us compare a high end CPU (Intel Xeon E5-2699 v3) 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 518 GFLOPS (double precision). The GPU has 1.5MB of cache. The CPU has 45MB of cache. The cache is growing every generation as well. The previous generation GPU (Tesla Fermi) had 768KB of cache. The previous Xeon(E7-8890) had 38MB 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 36 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.

2.3.1 ELLPACK

To enable better performance **?**) 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. The column values do not

matter in the padded zero cases so we mark them with don't care markers ($\times$).

$$\text{Matrix Data} = \begin{bmatrix} A_{11} & A_{14} & A_{17} & 0 \\ A_{25} & A_{28} & 0 & 0 \\ A_{32} & A_{33} & A_{36} & A_{37} \\ A_{41} & A_{45} & 0 & 0 \\ A_{53} & A_{54} & A_{57} & A_{58} \\ A_{62} & A_{65} & 0 & 0 \\ A_{72} & A_{73} & A_{76} & A_{78} \\ A_{83} & A_{84} & A_{85} & A_{86} \end{bmatrix}, \text{Column Indices} = \begin{bmatrix} 0 & 3 & 6 & \times \\ 4 & 7 & \times & \times \\ 1 & 2 & 5 & 6 \\ 0 & 4 & \times & \times \\ 2 & 3 & 6 & 7 \\ 1 & 4 & \times & \times \\ 1 & 2 & 5 & 7 \\ 2 & 3 & 4 & 5 \end{bmatrix} \tag{2.3}$$

In a CSR implementation each thread computes one row of the matrix. However, the ELLPACK matrix is stored in column major order. This enables coalescing memory access. Coalescing 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, $\times$, 5, $\times$, 6, $\times$, 5, 4, $\times$, $\times$, 6, $\times$, 7, $\times$, 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}

Bell and Garland 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.

2.3.2 Block-ELLPACK

ELLPACK has seen a lot of variations in the research literature. We discuss one design that marries BSR with ELLPACK called BELLPACK [**?**)]. 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 subblocks in every set of 2 rows. The Block-ELLPACK format for the example in equation **??** is shown below:

$$\text{Matrix Data} = \begin{bmatrix} \begin{bmatrix} A_{14} & 0 \\ 0 & A_{25} \end{bmatrix} \begin{bmatrix} A_{17} & 0 \\ 0 & A_{28} \end{bmatrix} \\ \begin{bmatrix} 0 & A_{32} \\ A_{41} & 0 \end{bmatrix} \begin{bmatrix} 0 & A_{36} \\ A_{45} & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & A_{53} \\ A_{62} & 0 \end{bmatrix} \begin{bmatrix} A_{54} & 0 \\ 0 & A_{65} \end{bmatrix} \\ \begin{bmatrix} A_{72} & A_{73} \\ 0 & A_{83} \end{bmatrix} \begin{bmatrix} 0 & A_{76} \\ A_{85} & A_{86} \end{bmatrix} \end{bmatrix}, \text{Column Indices} = \begin{bmatrix} 3 & 6 \\ 0 & 4 \\ 1 & 3 \\ 1 & 4 \end{bmatrix} \tag{2.4}$$

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}

2.4 FPGA

Like GPUs, FPGAs compute very differently than CPUs. 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 **?**) provided a reasonable matrix matrix multiplication design 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, consequently we believe 200 GFLOPS is achievable on this FPGA.

Several different HPC FPGA platforms exist. We use the Convey HC-2ex (Figure **??**). 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 is possible to design an SpMV processor to load on an FPGA and get competitive performance.

For example, memory or RAM blocks are distributed equally across the FPGA meaning that block RAMs can be located in a multiply-accumulator storing intermediate y values. In

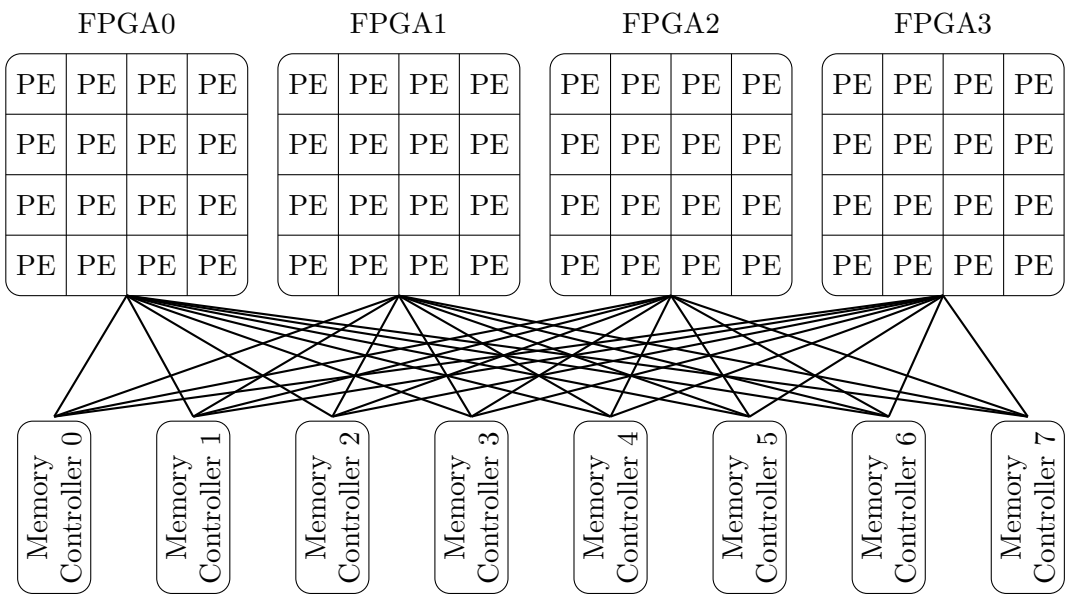


Figure 2.2: Most implementations on the Convey HC-2ex tile the 4 Xilinx Virtex-6 LX760 FPGAs with as many processing elements (PE) as possible. Each Virtex-6 chip connects to all 8 memory controllers, which enables each chip to have access to all of the coprocessor's memory.

a CPU the ALU (where the floating point computation takes place) has a fixed small amount of memory (registers) and access to more memory (the cache) is far away and has a higher latency making it less practical to store many intermediate y values.

2.4.1 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 CPUs play this way they have to lose some pipeline efficiency.

2.4.2 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 **??**). **?**) introduces delta compression for CPUs. This method stores the delta values in either 8, 16, 32, or 64 bits. However, we could use completely variable length codes to store deltas. On CPUs 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. Using Elias gamma coding [**?**)] to encode (row major traversal) deltas for the example in equation **??** is shown:
COMPRESSED ROW: 3, 5, 9, 11, 15, 17, 21, 25
DELTAS: 1, 3, 3, 5, 3, 2, 1, 3, 1, 1, 4, 3, 1, 3, 1, 2, 3, 2, 1, 3, 2, 3, 1, 1, 1
GAMMA CODES: 1, 011, 011, 00101, 011, 010, 0, 011, 0, 0, 00100, 011, 0, 011, 0, 010, 011, 010, 0, 011, 010, 011, 0, 0, 0

2.4.3 Value Compression

The same paper [**?**)] also uses value compression, for the matrix values. The idea is to take advantage of the fact values repeat. The unique values are stored in one array. Then a second array stores the indices into this array. This is beneficial when the unique values can fit in the L1 or L2 (lowest level) caches. Again, FPGAs can dedicate area for the decoder, and can potentially get better speedup results. Figure **??** shows the effect of easily compressed values on the performance of our previous work R^3 [**?**)].

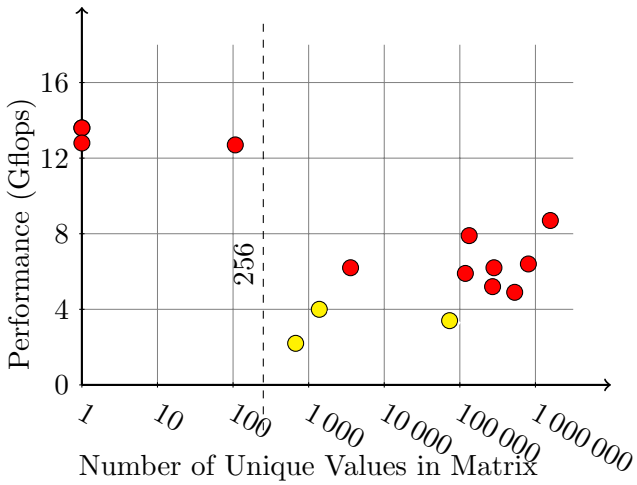


Figure 2.3: Unique values in a matrix vs the performance of R^3. Matrices with fewer than 256 unique values (only common elements exist) enables R^3 format to compress much better. The ●'s are outliers due to their size (see Figure **??**).

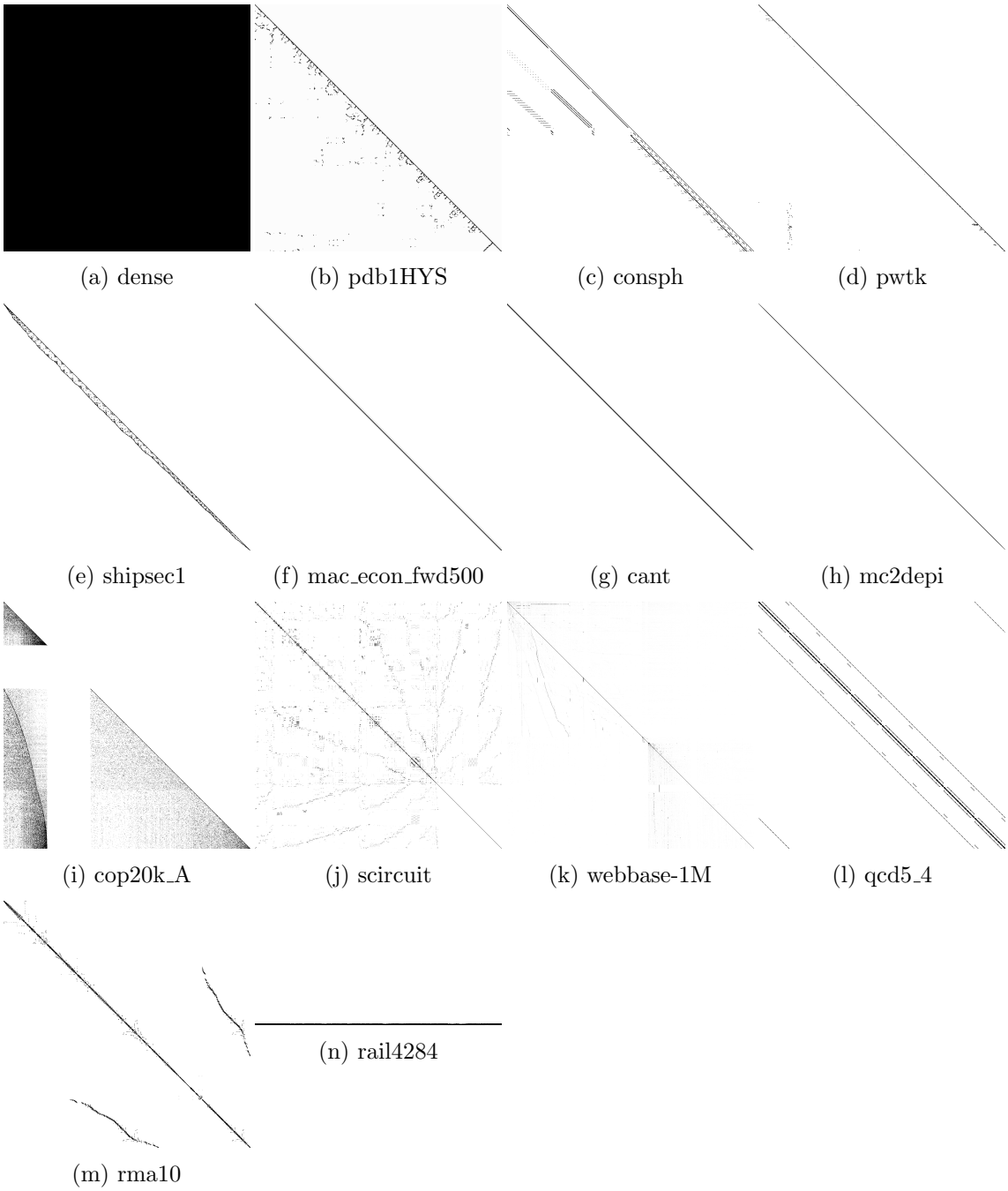
2.5 Benchmarking

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 where benchmarking comes in. However, different matrices can have vastly different SpMV performance. So a test set of matrices is used (Figure **??**). In Figure **??** 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 **??** 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



(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

(n) rail4284

(m) rma10

Figure 2.4: The density plots of the matrices used for testing

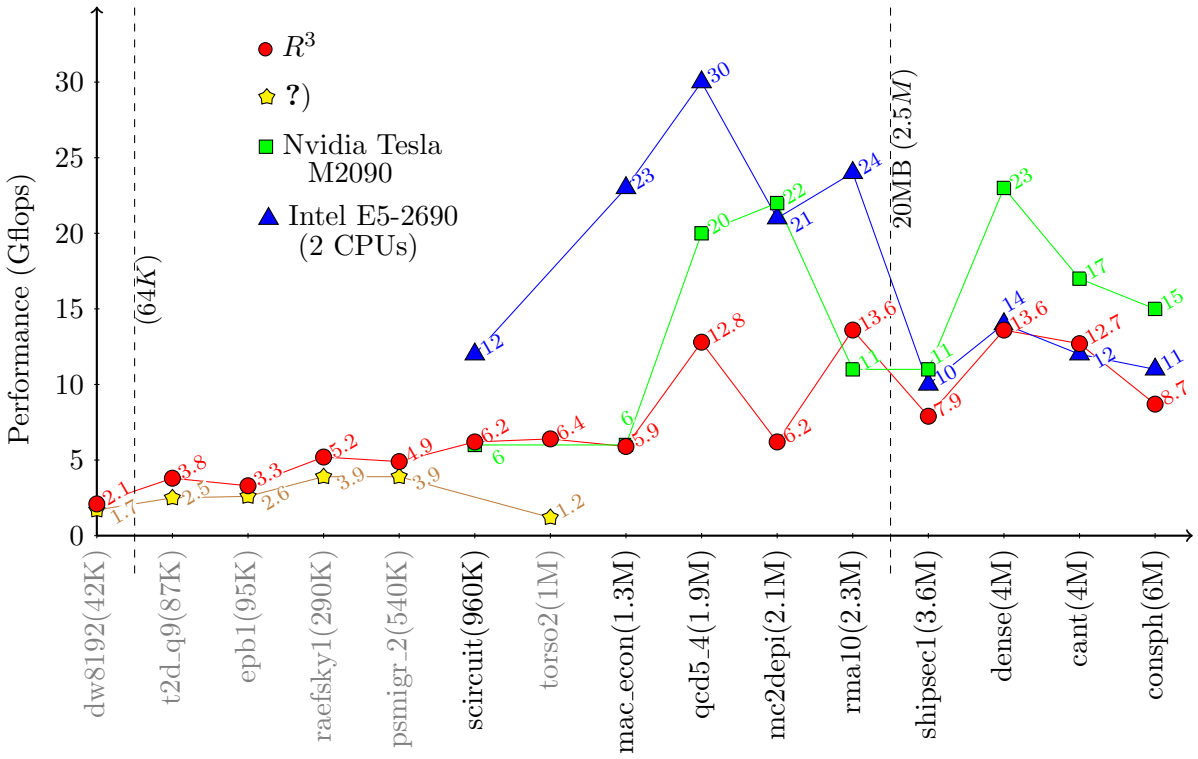


Figure 2.5: *nnz* vs Performance on each platform. The small matrices, ones around 64K or less, performed poorly on R^3, due to the overhead. CPUs experience the opposite effect. They take a performance hit once the matrix no longer fits in cache.

the most memory space. GPU boards currently have at most 12GB of on board RAM compared to the 128GB 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 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 discussed how others approach computing SpMV on FPGAs and other processors. We build upon these ideas and add our own. Three pillars emerged during the design of the hardware description and software: designing the traversal of the matrix, designing the multiply-accumulator, and designing the matrix compression. Figure **??** abstractly illustrates our view that all three pillars need to be in place to achieve competitive performance. The next three sections describe these pillars and the interactions between them.

3.1 First Pillar: Matrix Traversal

The first pillar, matrix traversal, primarily helps with vector reuse. Column traversal has a major effect on vector reuse. Many papers argue that vector caching is the way to achieve x 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 x vector caching.

The second advantage of mixing row and column traversal is that it leads to smaller deltas. In this paper, a delta is the traversal distance between a matrix element and its preceding matrix element in the traversal. To achieve high x vector reuse and small deltas we use row-column-row traversal. Chapter **??** discusses matrix traversal in detail.

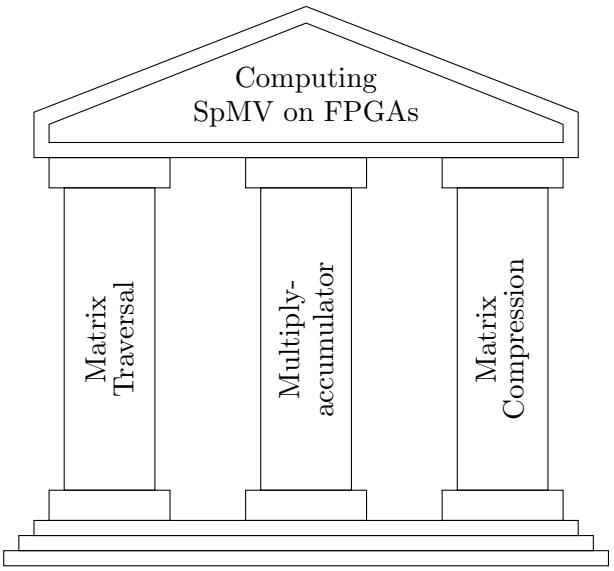


Figure 3.1: Because different aspects limit the performance of SpMV on FPGAs, no one optimization will lead to significant benefit for SpMV. We identified these three optimizations that together lead to a significant performance benefit.

3.2 Second Pillar: Multiply-accumulator

The second pillar, the multiply accumulator, has to accumulate multiple rows at a time to allow different traversals (the first pillar). Several multiply-accumulators exists, but they rely on row-major traversal. Although, we used pre-existing floating-point cores created by Flopoco [**?**)]. We created an IP core called an Intermediator, which stores intermediate y values and allows for row-column-row traversal. Chapter **??** discusses the multiply-accumulator in detail.

3.3 Third Pillar: Matrix Compression

The third pillar, compression, 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. Matrix compression is split into to separate problems: index compression and floating point compression.

3.3.1 Index Compression

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. Chapter **??** discusses delta compression in detail.

3.3.2 Floating Point Compression

Floating point 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 **??** in the previous chapter showed how much of an effect this pattern has on the performance of our previous SpMV implementation, R^3. Chapter **??** discusses our new floating point compression in detail.

3.3.3 Multi-port Shared Memory

Because good value compression requires a significant amount of on-chip memory space, we designed a shared memory IP block. This means instead of using 1 RAM block on each PE to store the 512 most common floating point values, we use one RAM block per PE to create a large shared memory to store the 8,192 most common floating point values. Chapter **??** discusses the design of the shared memory IP block.

3.4 High Level Design

Designing high performance reconfigurable computing implementations has a general two step process, which we follow. First, design one processing element (PE) to solve the problem (see Figure **??**). Second, replicate that PE until all the FPGAs are full (see Figure **??**). In addition to the PEs, we have a shared memory on the FPGA as well.

The PEs receive instructions through a 1D systolic array. Each PE has 14 registers. Each PE has 8 instructions: RESET, READ_REGISTER, WRITE_REGISTER, WRITE_DELTA_CODES,

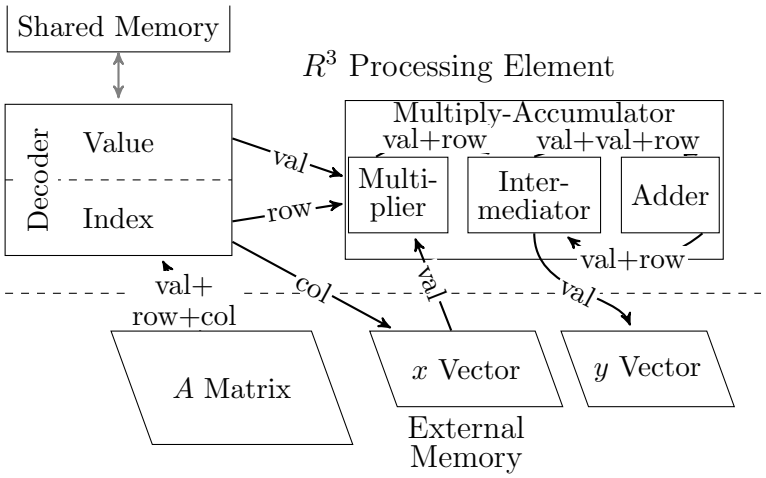


Figure 3.2: A single processing element. The arrows show the flow of data through the processing element. Although this diagram shows the memory access to each of the 3 places in memory as separate, they share one memory port. The diagram also does not show the FIFOs that help keep the pipeline full.

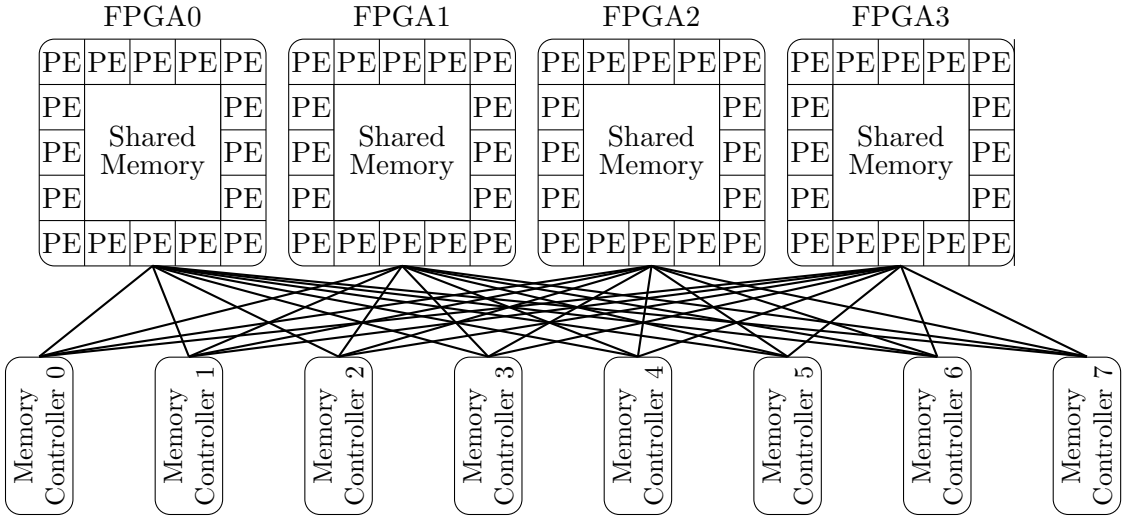


Figure 3.3: Our implementation has one shared memory for storing repeating values in the sparse matrices.

WRITE_FZIP_CODES, WRITE_SHARED_MEMORY, and COMPUTE_SPMV.

The PE themselves have 2 major components. First, the decoder block that requests the compressed matrix data and decodes it into row and column indices and floating point values. Second, the multiply-accumulator, where the actual SpMV computation takes place.

To Parallelize the SpMV computation we split the problem into N smaller SpMV computations each given to one processing element. In hardware we connect each processing element to an external memory port and a shared memory port. Chapter **??** discusses the high level design in detail.

CHAPTER 4. MATRIX TRAVERSAL

It may seem strange for matrix traversal to have its own chapter. However, it plays a big role in SpMV performance. Primarily, it effects the amount of x vector reuse the FPGA achieves. Secondarily, it effects the compression of indices. This chapter is organized as follows. Section **??** discusses traversals found in other work. Section **??** discusses our row column row (RCR) traversal and analyzes its performance. Section **??** discusses our design of the hardware to reuse x vector values when using RCR traversal. Section **??** discusses our choice for the subwidth and subheight parameters used in RCR traversal and discusses x vector reuse when cache lines need consideration.

4.1 Related Work

Most of the work on SpMV using FPGAs use a row-major traversal [**???**)]. However, we found two alternate traversals. The first does column traversal and caches the y vector. The second does something similar to ELLPACK, storing the matrix in column-major order but processing it in row-major order.

First, The work in **??**) does column-major traversal primarily to avoid caching the x vector. The secondary reason for column traversal is that makes the accumulator is only able to process unique rows in any stage in it's pipeline, however that is not relevant to this section. The downside of this approach is that it requires a y vector cache. This work focuses on configuring the cache to be large enough to prevent cache misses. However, this necessarily means that larger matrices will see worse performance. So this cache will not be particularly efficient for matrices with heights of 100,000 or more.

Second, The work in **?**) does a traversal similar to ELLPACK to make designing the FPGA

easier. The design is a mix between ELLPACK and the column row traversal mentioned in Section **??**. The traversal for the example in Equation **??** follows. In this example we set the subheight to 4. We also mark padded values as '×'.

VALUES: A_{11}, A_{25}, A_{32}, A_{41}, A_{14}, A_{28}, A_{33}, A_{45}, A_{17}, A_{62}, A_{36}, A_{83}, A_{53}, A_{65}, A_{37}, A_{84}, A_{54}, ×, A_{72}, A_{85}, A_{57}, ×, A_{73}, A_{86}, A_{58}, ×, A_{76}, ×, ×, ×, A_{78}, × The paper does not suggest this helps with x vector reuse. In fact, this paper limits itself to only matrices where the width (N) is small enough so that the entire x vector can fit on the FPGA.

Our previous work using column row traversal (from Section **??**) does quite well. In this case, if we want better x vector reuse we increase the subheight. However, as we increase the subheight the values become less 'clumped' and we get worse index compression.

4.2 Row Column Row (RCR) Traversal

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. To illustrate, let us look at the traversal in the example matrix from Chapter **??** in Equation **??**. In the following RCR traversal we set the subheight and subwidth parameters to 4 and 2 respectfully:

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}

So now that we know we are going to use RCR traversal, what are the optimal values for the subheight and subwidth? There are two metrics to look at. First, how much x vector reuse is achieved. Second, how much compression is improved.

4.2.1 x Vector Reuse

To analyze the x vector reuse, we look at two pieces of information. First, how many times on average the x vector values get used when they get fetched. Second, how many more times x vector values get fetched than if they were perfectly cached (only N fetches). Table **??** and Table **??** show these respective analyses. We use the benchmark set of matrices from Chapter **??**, Section **??** for our analysis.

Table 4.1: The average about of x vector reuse per x vector fetch from external memory for different matrices in the benchmark set.

	Subheight										
Matrix	1	2	4	8	16	32	64	128	256	512	1024
cant	1.0	1.7	2.8	4.1	5.3	7.9	11.5	15.0	20.7	31.3	42.2
consph	1.0	1.7	2.9	4.2	5.6	6.6	9.0	12.8	16.3	19.0	22.8
cop20k_A	1.0	1.6	1.9	2.1	2.3	2.5	2.6	2.7	2.8	2.9	3.1
dense2	1.0	2.0	4.0	8.0	16.0	31.7	62.5	125	250	500	1000
mac_econ_fwd500	1.0	1.1	1.2	1.4	1.6	1.8	2.2	2.6	2.9	3.5	4.4
mc2depi	1.0	1.1	1.2	1.3	1.3	1.3	1.3	1.3	1.4	1.7	2.4
pdb1HYS	1.0	1.9	3.2	5.4	8.7	13.5	19.7	26.7	33.8	40.8	49.0
pwtk	1.0	1.8	3.3	5.5	8.3	11.3	13.7	15.5	16.7	21.6	29.6
qcd5_4	1.0	1.5	2.2	2.9	3.9	4.8	5.5	6.8	8.0	8.8	11.1
rail4284	1.0	1.2	1.3	1.3	1.4	1.5	1.6	1.7	2.0	2.4	3.4
rma10	1.0	1.8	2.9	4.6	6.8	9.1	11.5	14.5	18.7	23.3	27.7
scircuit	1.0	1.3	1.5	1.7	1.8	2.0	2.1	2.2	2.3	2.4	2.5
shipsec1	1.0	2.0	3.1	4.3	5.5	6.8	8.2	9.7	11.2	13.0	15.0
webbase-1M	1.0	1.3	1.6	1.9	2.1	2.2	2.3	2.3	2.4	2.4	2.4

Table 4.2: The ratio of x vector fetches to the minimum possible (N) for different matrices in the benchmark set.

	Subheight										
Matrix	1	2	4	8	16	32	64	128	256	512	1024
cant	64.2	38.2	23.3	15.8	12.0	8.1	5.6	4.3	3.1	2.0	1.5
consph	72.1	41.3	25.2	17.0	12.9	10.9	8.0	5.6	4.4	3.8	3.2
cop20k_A	21.7	13.6	11.4	10.1	9.2	8.7	8.3	8.0	7.7	7.4	6.9
dense2	2000	1000	500	250	125	63.0	32.0	16.0	8.0	4.0	2.0
mac_econ_fwd500	6.2	5.6	5.1	4.5	3.9	3.4	2.8	2.4	2.1	1.8	1.4
mc2depi	4.0	3.5	3.2	3.1	3.1	3.0	3.0	3.0	2.9	2.4	1.7
pdb1HYS	119.3	64.4	37.1	22.1	13.7	8.8	6.1	4.5	3.5	2.9	2.4
pwtk	53.4	29.7	16.4	9.8	6.4	4.7	3.9	3.4	3.2	2.5	1.8
qcd5_4	39.0	26.0	17.5	13.2	10.1	8.1	7.0	5.7	4.9	4.4	3.5
rail4284	10.3	8.9	8.2	7.7	7.3	6.9	6.4	5.9	5.2	4.3	3.0
rma10	50.7	28.9	17.4	10.9	7.5	5.6	4.4	3.5	2.7	2.2	1.8
scircuit	5.6	4.5	3.8	3.4	3.1	2.8	2.7	2.6	2.4	2.3	2.2
shipsec1	55.5	27.7	17.8	12.9	10.1	8.2	6.8	5.7	4.9	4.3	3.7
webbase-1M	3.1	2.3	1.9	1.7	1.5	1.4	1.4	1.3	1.3	1.3	1.3

Table 4.3: This table shows the effect of changing the subwidth and subheight of RCR traversal on the estimated compression size. Each measurement is the average estimated bits per delta over all the benchmark matrices.

	Subwidth										
Subheight	1	2	4	8	16	32	64	128	256	512	1024
1	3.26	3.26	3.26	3.26	3.26	3.26	3.26	3.26	3.26	3.26	3.26
2	2.30	2.34	2.41	2.49	2.57	2.64	2.72	2.79	2.86	2.92	2.97
4	1.92	1.92	1.98	2.09	2.20	2.31	2.42	2.53	2.63	2.72	2.81
8	1.84	1.75	1.78	1.88	2.00	2.12	2.25	2.38	2.50	2.61	2.71
16	1.88	1.70	1.69	1.77	1.89	2.01	2.15	2.29	2.43	2.55	2.66
32	1.98	1.71	1.66	1.73	1.84	1.96	2.09	2.24	2.39	2.51	2.62
64	2.12	1.76	1.66	1.71	1.81	1.93	2.07	2.21	2.36	2.49	2.61
128	2.27	1.82	1.69	1.71	1.80	1.92	2.05	2.20	2.35	2.48	2.60
256	2.42	1.89	1.72	1.72	1.80	1.91	2.04	2.19	2.34	2.48	2.59
512	2.55	1.95	1.74	1.73	1.81	1.91	2.04	2.19	2.34	2.47	2.59
1024	2.66	2.00	1.77	1.74	1.81	1.91	2.04	2.19	2.34	2.47	2.59

Table 4.4: This table shows the effect of changing the subwidth and subheight of RCR traversal on the estimated compression of one of the most difficult to compress matrices in the benchmark, mac_econ_fwd500.

	Subwidth										
Subheight	1	2	4	8	16	32	64	128	256	512	1024
1	5.80	5.80	5.80	5.80	5.80	5.80	5.80	5.80	5.80	5.80	5.80
2	4.99	5.03	5.06	5.10	5.15	5.19	5.24	5.28	5.33	5.36	5.42
4	4.55	4.63	4.67	4.72	4.79	4.84	4.91	4.98	5.06	5.11	5.20
8	4.32	4.41	4.46	4.53	4.61	4.68	4.75	4.83	4.92	4.99	5.08
16	4.25	4.33	4.37	4.43	4.52	4.58	4.65	4.74	4.85	4.93	5.02
32	4.32	4.37	4.40	4.44	4.51	4.55	4.61	4.71	4.82	4.90	4.98
64	4.38	4.40	4.40	4.42	4.48	4.51	4.58	4.68	4.80	4.88	4.97
128	4.46	4.42	4.40	4.41	4.46	4.49	4.55	4.66	4.78	4.87	4.96
256	4.60	4.49	4.43	4.42	4.47	4.48	4.54	4.64	4.78	4.87	4.95
512	4.69	4.52	4.45	4.43	4.47	4.48	4.54	4.64	4.78	4.86	4.95
1024	4.79	4.56	4.46	4.43	4.47	4.48	4.54	4.64	4.77	4.86	4.95

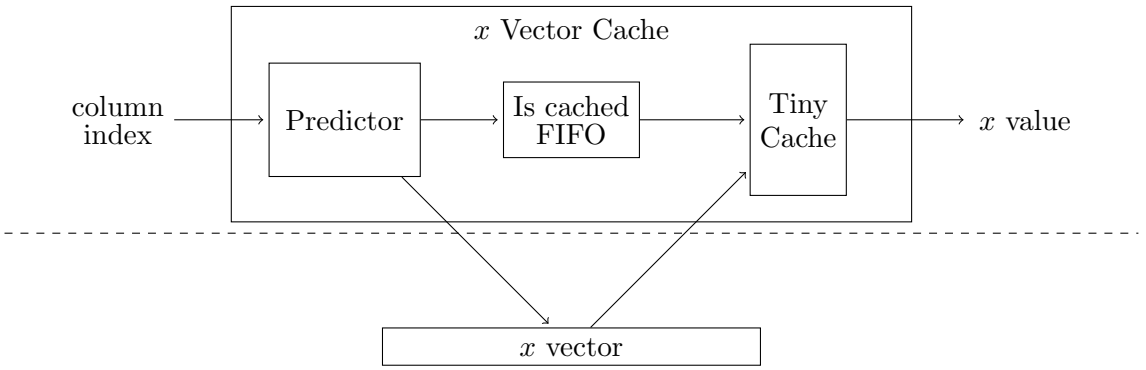


Figure 4.1: The x vector cache block is responsible for turning column indices into x vector values.

4.2.2 Improving Index Compression

As we mentioned traversal also affects index compression. This is because the traversal can make the deltas more or less 'clumpy'. In other words, it is easier to compress many small deltas and a large delta rather than many medium deltas. To estimate the compression, we can use the formula $\sum_{i=1}^{nnz} \log_2(d_i + 1)$, where d is the list of delta values. This formula approximately equals the sum of the bits required to represent each delta. We collected the estimated compression for each matrix for subwidths and subheights ranging from 1 to 1024 by powers of 2. This generates too much data to show in this paper, so we show the averages across all the matrices in Table **??**. We also show the most difficult to compress (in terms of its indices) in Table **??**.

4.3 x Vector Cache

In order to achieve the amount of reuse in Tables **??** and **??**, we need to cache sw x vector values, where sw is equal to the subwidth we use. This may seem hypocritical since we advocated traversal as an alternative to caching to achieve x vector reuse. However, the number of cached values is small and the hardware to cache these values is simple and requires a small area.

Figure **??** shows our design for the x Vector Cache. Our x Vector Cache consists of two main parts. First, a Predictor predicts whether the to be requested x vector value will be in the cache. Second, a Tiny Cache stores x vector values.

Table 4.5: Example operation of the x vector cache.

Column Index (Input)	Predictor State	Prediction	Read x Vector Value	Updated Cache	Output
1	0000	Not Cached	x_1	$x_1, \times, \times, \times$	x_1
4	1000	Not Cached	x_4	$x_1, \times, \times, x_4$	x_4
2	1001	Not Cached	x_2	$x_1, x_2, \times, x_4$	x_2
3	1101	Not Cached	x_3	x_1, x_2, x_3, x_4	x_3
1	1111	Cached	None	x_1, x_2, x_3, x_4	x_1
7	0000	Not Cached	x_7	x_1, x_2, x_7, x_4	x_7
5	0010	Not Cached	x_5	x_5, x_2, x_7, x_4	x_5
8	1010	Not Cached	x_8	x_5, x_2, x_7, x_8	x_8
6	1011	Not Cached	x_6	x_5, x_6, x_7, x_8	x_6
7	1111	Cached	None	x_5, x_6, x_7, x_8	x_7
5	1111	Cached	None	x_5, x_6, x_7, x_8	x_5
3	0000	Not Cached	x_3	x_5, x_6, x_3, x_8	x_3
4	0010	Not Cached	x_4	x_5, x_6, x_3, x_4	x_4
2	0011	Not Cached	x_2	x_5, x_2, x_3, x_4	x_2
2	0111	Cached	None	x_5, x_2, x_3, x_4	x_2
3	0111	Cached	None	x_5, x_2, x_3, x_4	x_3
3	0111	Cached	None	x_5, x_2, x_3, x_4	x_3
4	0111	Cached	None	x_5, x_2, x_3, x_4	x_4
7	0000	Not Cached	x_7	x_5, x_2, x_7, x_4	x_7
8	0010	Not Cached	x_8	x_5, x_2, x_7, x_8	x_8
5	0011	Not Cached	x_5	x_5, x_2, x_7, x_8	x_5
6	1011	Not Cached	x_6	x_5, x_6, x_7, x_8	x_6
8	1111	Cached	None	x_5, x_6, x_7, x_8	x_8
5	1111	Cached	None	x_5, x_6, x_7, x_8	x_5
6	1111	Cached	None	x_5, x_6, x_7, x_8	x_6

Table **??** shows the operation of the x Vector Cache, when the example matrix in Equation **??** (from Chapter **??**). In this example we set sw (subwidth) to 4. The Predictor has an array of 4 bits to predict if a value will be cached. The Predictor uses the column mod subwidth index to predict if the value will be in cache. After the colmun index is processed the corresponding bit (in the Predictor) is changed to 1. When the column indices move to a new column grouping (for example when the sixth column index arrives) the bits reset to 0. Based on the prediction the associated x value is requested or no value is requested.

Since the values need to be outputted in order and some have to wait on external memory latency the predictions are put in a fifo. The values with a 'Not Cached' prediction wait for the next x vector value from external memory to update the cache and that value is outputted. The values with a 'Cached' prediction do not need to wait and query the corresponding location in the cache, which is sent to the output.

The x Vector Cache uses a small amount of area. Using xst and targeting the Xilinx Virtex-6 LX760, the design uses 346 LUTs, 280 registers and 1 RAM block. The design reaches an achievable frequency of 243Mhz. In this case we set the depth of the Is Cached FIFO to 1024. We also use a FIFO to store x vector responses with a depth of 512 (this is what uses the RAM block).

4.4 Results and Discussion

From our analysis in the previous sections we choose to use a subheight equal to 512 and a subwidth equal to 8. The Convey-HC2ex uses SG-DIMMS (scatter-gather DIMMs) to be able to access 8-byte values randomly and at near full throughput, however, most platforms do not support this and access 64 byte lines. We analyse the effect of reading 8 x vector values instead of one in Table **??**. As seen, 4 matrices end up requesting 50% or more values, but the other 10 use less than 50% more bandwidth.

Table 4.6: Many platforms need to access external memory in 64-byte cache lines. This results in a loss of performance compared to using SG-DIMMs where any series of 8-byte values can be requested.

Matrices	SG-DIMMs		Regular		Percent more requests
	Reuse	ratio to (N)	Reuse	ratio to (N)	
cant	31.3	2.0	31.0	2.1	1.1%
consph	19.0	3.8	16.3	4.4	16.4
cop20k_A	2.9	7.4	1.0	20.7	180.5
dense2	500.0	4.0	500.0	4.0	0.0
mac_econ_fwd500	3.5	1.8	1.8	3.5	99.2
mc2depi	1.7	2.4	1.7	2.4	1.1
pdb1HYS	40.8	2.9	34.0	3.5	20.0
pwtk	21.6	2.5	20.9	2.6	3.6
qcd5_4	8.8	4.4	7.1	5.5	23.3
rail4284	2.4	4.3	1.7	6.2	43.8
rma10	23.3	2.2	20.8	2.4	12.1
scircuit	2.4	2.3	1.0	5.4	129.3
shipsec1	13.0	4.3	11.7	4.7	10.7
webbase-1M	2.4	1.3	1.4	2.3	74.8

CHAPTER 5. MULTIPLY-ACCUMULATOR

A high throughput SpMV implementation relies on designing a rarely stalling multiply accumulator (MAC). An inefficient engine will often stall when a matrix and its associated vector value arrives every or nearly every clock cycle. The long latency of floating point addition and row-column-row traversal makes this complicated. This chapter is organized as follows. Section **??** discusses previous approaches to accumulator designs. Section **??** discusses our MAC, which uses an 'Intermediator' to manage intermediate y values. Section **??** discusses performance and synthesis results of our design.

5.1 Related Work

We looked at several multiply-accumulator designs before creating our own. We also use floating-point cores from Flopoco [**?**)], so we are not starting from scratch. There are many different floating-point multiply-accumulators that solve different problems. Most accept one matrix element and one x vector value at a time, however, we also discuss one that does not.

5.1.1 Floating Point Adder and Multiplier (Flopoco)

Most designs, including ours, use existing double-precision adder and multiplier cores for their multiply accumulator. We use Flopoco to build our adder and multiplier [**??**)]. We use the performance and latency (See Table **??**) from these cores for reference in the rest of this section. Flopoco uses a slightly different floating point format than the IEEE 754 standard. In addition to the floating-point adder and the floating-point multiplier we used 2 cores for converting to and from IEEE 754 format.

Table 5.1: Flopoco core information.

IP Core	LUTs	Registers	DSPs	Frequency	Pipeline Stages
Adder (FPAdder_11_52_uid2)	1,190	1,136	0	298Mhz	14
Multiplier (FPMultiplier_11_52_11_52_11_52_uid2)	1,067	1,061	7	314Mhz	11
IEEE 754 to Flopoco (InputIEEE_11_52)	67	68	0	280Mhz	1
Flopoco to IEEE 754 (OutputIEEE_11_52)	65	67	0	260Mhz	1

5.1.2 Strided Accumulator

One way to accumulate values with an adder that has a latency of α clock cycles is to accumulate α values at a time. This requires values to come in a strided manner. For example, once a value from row 0 enters the adder than $\alpha - 1$ more values from different rows enter the adder before proceeding to the next value in row 0. The traversal in **?**), discussed in the previous chapter in Section **??**, follows this rule and makes designing this accumulator fairly straight forward.

5.1.3 Binary Tree Accumulator

Generally one multiplier multiplicand pair is sent per clock cycle to a multiply-accumulator, however one design streams multiple values at a time [**?**)]. Effectively, this MAC can be used to create a large processing element rather than many smaller ones. The basic design is shown in Figure **??**. From a high level you can see this hardware computes dot-products. However, because the dot-products are of variable length the design is more complex.

This binary tree accumulator design uses an IPV (Input Pattern Vector) or a eol bitmap to keep track of the rows. A '1' indicates the element is the last element in the row and a '0' otherwise. Our example MAC accepts 4 pairs of values per clock cycle. To explain this MAC let us use the example matrix from Equation **??**. The inputs in this case would be as follows:

1. IPV=0010, data=$(A_{11} \times x_1, A_{14} \times x_4, A_{17} \times x_7, A_{25} \times x_5)$

2. IPV=1000, data=$(A_{28} \times x_8, A_{32} \times x_2, A_{33} \times x_3, A_{36} \times x_6)$

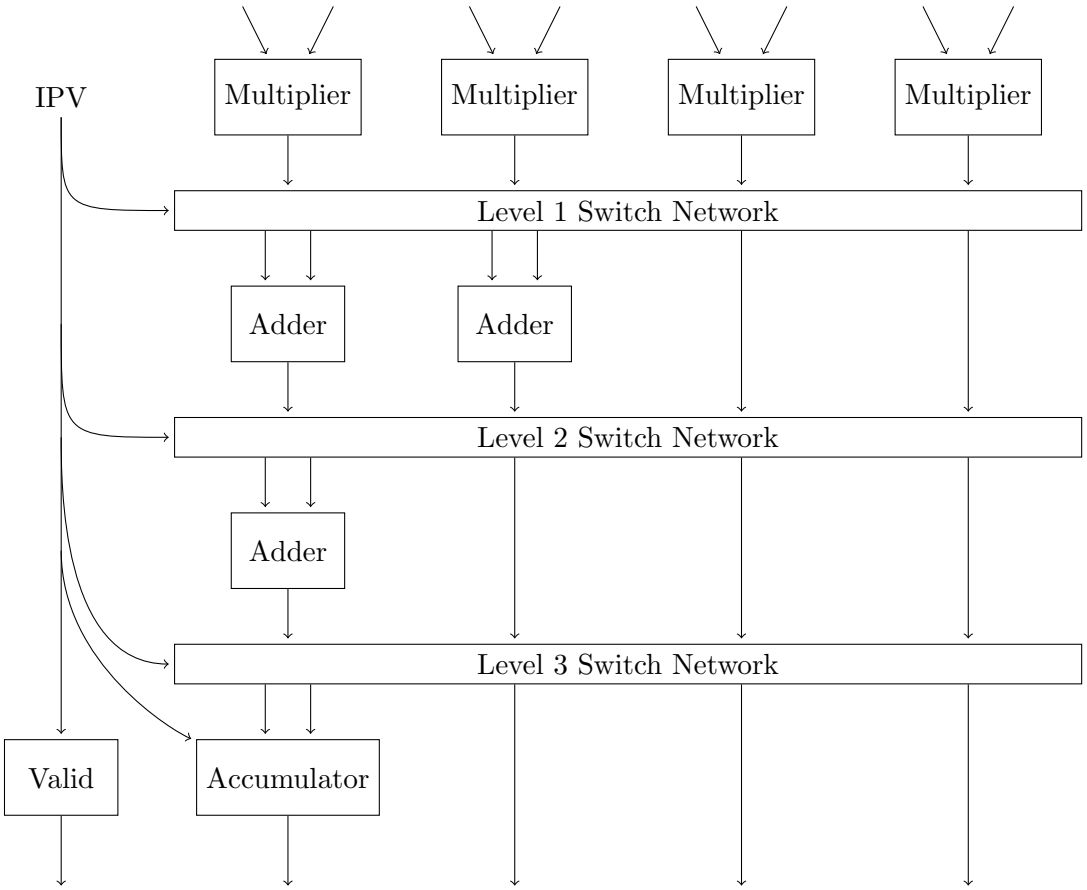


Figure 5.1: The binary tree accumulator.

3. IPV=1010, data=$(A_{37} \times x_7, A_{41} \times x_1, A_{45} \times x_5, A_{53} \times x_3)$

4. IPV=0010, data=$(A_{54} \times x_4, A_{57} \times x_7, A_{58} \times x_8, A_{62} \times x_2)$

5. IPV=1000, data=$(A_{65} \times x_5, A_{72} \times x_2, A_{73} \times x_3, A_{76} \times x_6)$

6. IPV=1001, data=$(A_{78} \times x_8, A_{83} \times x_3, A_{84} \times x_4, A_{86} \times x_6)$

The IPV controls 3 things: the routing in the switch networks, how the accumulator operates and how many outputs are valid. Table **??** shows one possible configuration of the IPVs for the cases that appear in the example.

You may notice that this computation does the additions out of order. 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 (in general) compute floating point addition in order (eg. $((1 + 2) + 3) + 4$). This means results

Table 5.2: Operation of the binary tree accumulator based on the Input Pattern Vector (IPV).

IPV	0010	1000	1010	1001
Level 1 Switch Network	$IN_0 \rightarrow OUT_0$	$IN_0 \rightarrow OUT_0$	$IN_0 \rightarrow OUT_0$	$IN_0 \rightarrow OUT_0$
	$IN_1 \rightarrow OUT_1$	$0 \rightarrow OUT_1$	$0 \rightarrow OUT_1$	$0 \rightarrow OUT_1$
	$IN_2 \rightarrow OUT_2$	$IN_1 \rightarrow OUT_2$	$IN_1 \rightarrow OUT_2$	$IN_1 \rightarrow OUT_2$
	$0 \rightarrow OUT_3$	$IN_2 \rightarrow OUT_3$	$IN_2 \rightarrow OUT_3$	$IN_2 \rightarrow OUT_3$
	$IN_3 \rightarrow OUT_4$	$IN_3 \rightarrow OUT_4$	$IN_3 \rightarrow OUT_4$	$IN_3 \rightarrow OUT_4$
	$\times \rightarrow OUT_5$	$\times \rightarrow OUT_5$	$\times \rightarrow OUT_5$	$\times \rightarrow OUT_5$
Level 2 Switch Network	$IN_0 \rightarrow OUT_0$	$IN_1 \rightarrow OUT_0$	$IN_0 \rightarrow OUT_0$	$IN_1 \rightarrow OUT_0$
	$IN_1 \rightarrow OUT_1$	$IN_2 \rightarrow OUT_1$	$0 \rightarrow OUT_1$	$IN_2 \rightarrow OUT_1$
	$IN_2 \rightarrow OUT_2$	$IN_0 \rightarrow OUT_2$	$IN_1 \rightarrow OUT_2$	$IN_0 \rightarrow OUT_2$
	$\times \rightarrow OUT_3$	$\times \rightarrow OUT_3$	$IN_2 \rightarrow OUT_3$	$\times \rightarrow OUT_3$
	$\times \rightarrow OUT_4$	$\times \rightarrow OUT_4$	$\times \rightarrow OUT_4$	$\times \rightarrow OUT_4$
Level 3 Switch Network	$IN_1 \rightarrow OUT_0$	$IN_0 \rightarrow OUT_0$	$IN_2 \rightarrow OUT_0$	$0 \rightarrow OUT_0$
	$IN_0 \rightarrow OUT_1$	$IN_1 \rightarrow OUT_1$	$IN_0 \rightarrow OUT_1$	$IN_1 \rightarrow OUT_1$
	$\times \rightarrow OUT_2$	$\times \rightarrow OUT_2$	$IN_1 \rightarrow OUT_2$	$IN_0 \rightarrow OUT_2$
	$\times \rightarrow OUT_3$	$\times \rightarrow OUT_3$	$\times \rightarrow OUT_3$	$\times \rightarrow OUT_3$
	$\times \rightarrow OUT_4$	$\times \rightarrow OUT_4$	$\times \rightarrow OUT_4$	$\times \rightarrow OUT_4$
valid	1000	1000	1100	1100

may differ slightly, because changing the order of floating point addition can change the result [**?**)].

The accumulator has 2 inputs. The right input is for the sum the represents the first row of the current set that may be a continuation of the last row in the previous set. The left input is for storing the part of the dot product that continues onto the next input set. In the case that no row end in the current set (IPV=0000) then the value goes into the right input of the accumulator.

5.1.4 The Log Sum Accumulator

One accumulator that would work for this design is a log sum accumulator [**?**)]. To allow this accumulator to be used in the previous accumulator, we have two inputs into this accumulator.

This design is one of the simplest accumulators. First, one floating point adder loops back on itself to accumulating new values with the adders output. Second, once the last value arrives the 14 or fewer values in the adder are sent through a series of adders. Each adder in the series reduces the number of values in the set by half. So 4 adders will can reduce up to 16 values,

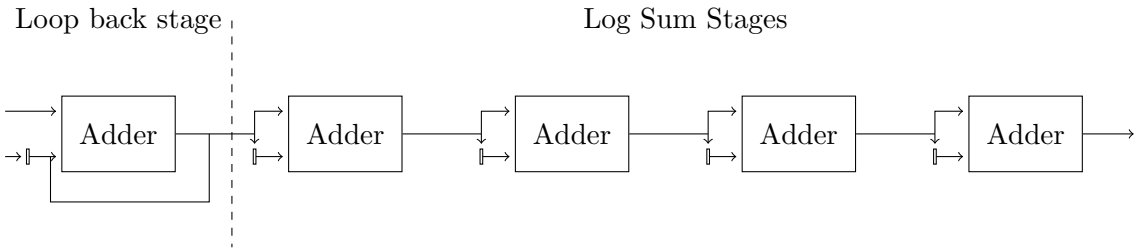


Figure 5.2: The log sum accumulator.

which is more than the 14 possible, down to 1 value.

The disadvantage of this design is that it uses 5 floating point adders, when on average less than 1 addition occurs each clock cycle. This means this design uses at least 6,000 LUTs and 5,700 registers. However, more efficient, but more complex designs exist.

5.1.5 Single Adder Accumulator

A multiply-accumulator with one multiplier should only need one adder to do all the accumulation. The difficulty is how to create an algorithm to achieve this. It is clear that buffering has to occur to achieve this. If the output needs to be in order than the buffering needs to be at least $O(\alpha log \alpha)$ to accommodate the case where one row has several values and fills the adder pipeline and the following rows have only 1 value. If the output can be out of order (meaning the output of the short rows that get accumulated quickly can go to the output before the long rows before it) than $O(\alpha)$ buffering may be possible.

Other work has achieved accumulators with only 1 or 2 adders. **?**) came up with 2 accumulators that use 2 adders and 1 that uses a single adder. However, the single adder solution had restrictions. In addition to only allowing row-major traversal it had a maximum row length of α (the depth of the adder pipeline). This design also required $O(\alpha^2)$ buffer space. This accumulator has two buffers. While one buffer is having its values accumulated the other is receiving new values. The adder takes in two values from the buffer in the accumulation state and then jumps α addresses to add two other values that belong to a different row. Table **??** shows the scheduling of accumulating the products from the example in Equation **??** back in Chapter **??**. For simplification we set $\alpha = 4$ instead of $\alpha = 14$.

Some may notice a hazard occurs if two value in the fourth column of the buffer belong

to the same row of the matrix. This is what prevents this accumulator from being able to accumulate rows greater than α. To accommodate this case the accumulator could have an additional adder or stall the accumulator to allow time for accumulating the longer rows.

5.1.6 Accumulator Using a Single Cycle Accumulation Loop

Another solution to this problem is to accumulate in one clock cycle. This is the approach in [**?**)]. The idea is to split the accumulation into 3 phases: The extend mantissa and reduce exponent phase, the accumulation loop phase, and the normalize phase (See Figure **??**). In the first phase the floating point format is converted into a 5-bit exponent and a 120-bit 2s complement mantissa. In the second phase the incoming value is added to the stored value. In the third phase, the stored value is converted back into IEEE-754 format.

Values come out of the first phase in the following format: $m_i \times 64^{e_i}$. One way to look at this is converting from base-2 scientific notation to base-64 scientific notation. The accumulation phase is more complex than a simple 2s complement adder, because of the exponent. The difference in the exponents determines what operation the accumulator does. If the incoming exponent (e_i) is more than 1 greater than the stored exponent (e_s) than the stored value is thrown out because it is too small and the incoming value replaces the stored value. The other cases are in Table **??**.

There is a hazard of overflowing or underflow that must be taken into account. By monitoring when there is a risk of overflowing (the first 2 bits of the stored mantissa are different) or underflowing the (first 66 bits are the same) then the result needs to be shifted right or left 64 bits to avoid the hazard.

As you may be noticing even this method is hampering the max frequency. Phase two needs a leading 0/1 detector to determine if m_s is at risk of underflowing or overflowing. Simultaneously $e_s - e_i$ is being calculated. This then determines the addition to occur. Alternatively, all possible additions can occur first and then the correct value is muxed to the input of the m_s register. Using a Xilinx Virtex-5 the work in **?**) achieved a frequency of 134Mhz to 247Mhz depending on the design trade offs they used.

Table 5.3: Operation of a single adder accumulator.

Cycle	First buffer				Adder	Second buffer			
1-4	$A_{11}x_1$	$A_{14}x_4$	$A_{17}x_7$	$A_{25}x_5$					
	$A_{28}x_8$	$A_{32}x_2$	$A_{33}x_3$	$A_{36}x_6$					
	$A_{37}x_7$	$A_{41}x_1$	$A_{45}x_5$	$A_{53}x_3$					
	$A_{54}x_4$	$A_{57}x_7$	$A_{58}x_8$	$A_{62}x_2$					
5-8			$A_{17}x_7$	$A_{25}x_5$	$A_{11}x_1 + A_{14}x_4$	$A_{65}x_5$	$A_{72}x_2$	$A_{73}x_3$	$A_{76}x_6$
	$A_{28}x_8$	$A_{32}x_2$	$A_{33}x_3$	$A_{36}x_6$					
	$A_{37}x_7$	$A_{41}x_1$	$A_{45}x_5$	$A_{53}x_3$					
			$A_{58}x_8$	$A_{62}x_2$	$A_{54}x_4 + A_{57}x_7$				
9-12				$A_{25}x_5$	$(A_{11}x_1 + A_{14}x_4) + A_{17}x_7$	$A_{65}x_5$	$A_{72}x_2$	$A_{73}x_3$	$A_{76}x_6$
	$A_{28}x_8$			$A_{36}x_6$	$A_{32}x_2 + A_{33}x_3$	$A_{78}x_8$	$A_{83}x_3$	$A_{84}x_4$	$A_{85}x_5$
	$A_{37}x_7$			$A_{53}x_3$	$A_{41}x_1 + A_{45}x_5$				
				$A_{62}x_2$	$(A_{54}x_4 + A_{57}x_7) + A_{58}x_8$				
13-16				$A_{25}x_5$		$A_{65}x_5$	$A_{72}x_2$	$A_{73}x_3$	$A_{76}x_6$
	$A_{28}x_8$				$(A_{32}x_2 + A_{33}x_3) + A_{36}x_6$	$A_{78}x_8$	$A_{83}x_3$	$A_{84}x_4$	$A_{85}x_5$
	$A_{37}x_7$			$A_{53}x_3$		$A_{86}x_6$			
	$A_{54}x_4 + A_{57}x_7 + A_{58}x_8$				$A_{62}x_2$				
17-20						$A_{65}x_5$	$A_{72}x_2$	$A_{73}x_3$	$A_{76}x_6$
					$A_{25}x_5 + A_{28}x_8$	$A_{78}x_8$	$A_{83}x_3$	$A_{84}x_4$	$A_{85}x_5$
					$(A_{32}x_2 + A_{33}x_3 + A_{36}x_6) + A_{37}x_7$	$A_{86}x_6$			
					$A_{53}x_3 + (A_{54}x_4 + A_{57}x_7 + A_{58}x_8)$				

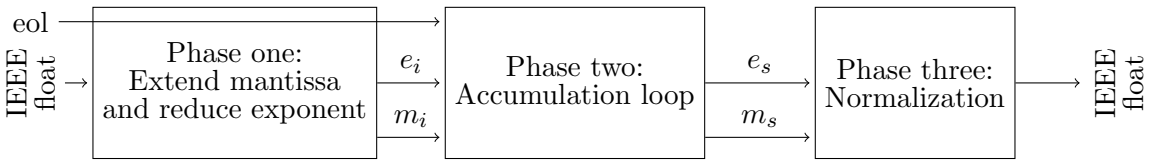


Figure 5.3: This shows the basic design of an accumulator with the goal of minimizing the critical loop path so that values can be accumulated in order.

Table 5.4: Based on the difference of the incoming exponent e_i and the stored exponent e_s the accumulator does a different operation to add the two values together

	underflow		overflow		other	
$e_s - e_i$	$e_s \Leftarrow$	$m_s \Leftarrow$	$e_s \Leftarrow$	$m_s \Leftarrow$	$e_s \Leftarrow$	$m_s \Leftarrow$
> 1	$e_s - 1$	$shl(m_s)$	$e_s + 1$	$shr(m_s)$	e_s	m_s
1	$e_s - 1$	$shl(m_s) + m_i$	$e_s + 1$	$shr(m_s)$	e_s	$m_s + shr(m_i)$
0	e_s	$m_s + m_i$	$e_s + 1$	$shr(m_s) + shr(m_i)$	e_s	$m_s + m_i$
-1	e_i	m_i	e_i	$m_i + shr(m_s)$	e_i	$m_i + shr(m_s)$
< -1	e_i	m_i	e_i	m_i	e_i	m_i

5.2 Multiply-accumulator with an Intermediator

The problem with all these designs is that they require row-major traversal. This handicap makes these MACs unusable for our purposes. The requirements of our MAC is as follows:

1. Within a 512 row section the MAC can receive elements in any order.
2. The MAC must receive all the elements in one 512 row section before proceeding to the next 512 row section.
3. Each row must have at least one element.
4. The MAC can stall, but not stall often enough to significantly effect throughput.

To achieve this we created a MAC that uses one multiplier and one adder and one Intermediator block (See Figure **??**). This Intermediator stores up to one intermediate y value for each row in a dual port RAM. This RAM is central to the design of the Intermediator.

In our previous work, R^3 [**?**)], we introduced the Intermediator but it was only capable of storing 32 intermediate y vector values. In this design, we expand this to 1024 (the minimum depth of one dual port RAM block in most Xilinx, Altera and Lattice FPGAs). Both Intermediator designs allow the matrix to be traversed in a loosely row major traversal and the MAC

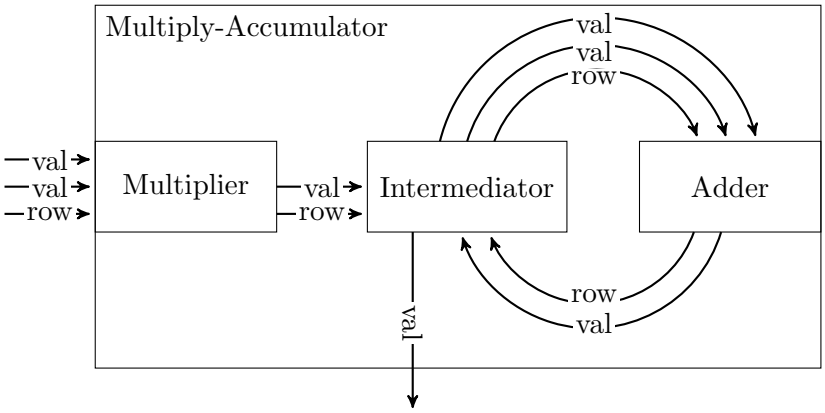


Figure 5.4: The multiply-accumulator block handles multiple intermediate y values at a time. Every clock cycle, the Intermediator block sends 2 values that belong to the same row to the adder, while simultaneously it receives 1 value from the adder and 1 from the multiplier, which my belong to different rows.

still behaves correctly. The step from 32 to 1024 intermediate values allows more freedom in the traversal. Earlier in Chapter **??**, we discussed traversals that abide by this rule and allow for easy reuse of x vector values.

5.2.1 Memory and States

The RAM block in the Intermediator has a depth of 1024. Each slot in memory has 2 types of states. First, each slot is either occupied or vacant. Second, each slot in either the red (active), yellow (fading), green (accumulated), or white (blank) states.

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 state and ready to be stored. Once the values have been sent out to be stored the memory transitions to the white blank state.

5.2.2 Operations

The Intermediator (Figure **??**) 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 **??**) stores intermediate values until an element in the same row appears.

Ideally the Intermediator receives a value from the multiplier and one value from the adder every clock cycle. These values often belong to different rows. The Intermediator also outputs one pair of values belonging to the same row to the adder every clock cycle.

So the Intermediator plays a game where it receives 2 values belonging to different rows and sends 2 values belonging to the same row.

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

Case 1: (Figure **??**) 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 **??**) 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 **??**) 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 **??**, **??**) 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 **??**) 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 **??**) Both values have row indexes that correspond to occupied cells in the

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 **??**) 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 **??** 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.

5.2.3 Stalls

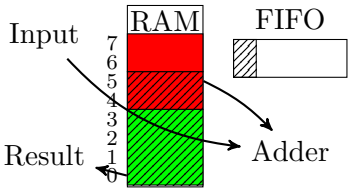
There are three hazards to consider when designing this MAC. First, if the intermediator-to-adder fifo is at risk of overflowing. Second, if the red window advances before the other half of the memory is entirely in the white (blank) state. Third, if the fading state does not last long enough for all the values to get accumulated.

For the first case, the intermediator-to-adder FIFO would be at risk of overflowing when case 6 (both inputs coorespond to full slots in the intermediator) occurs very often. Although we do not have a proof, we found no way for the intermediator-to-adder FIFO to ever get more than 8. But, in case this does happen, we designed the FIFO to signal a stall to prevent more values from coming in.

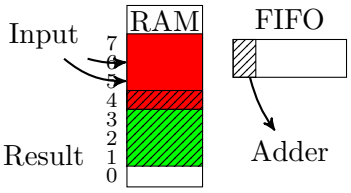
For the second case, the Intermediator would be at risk of receiving values from the multiplier that would overwrite values that are accumulated and being stored. This can happen if most of the rows have only one value.

A new FIFO between the multiplier and intermediator multiplier-to-intermediator FIFO is introduced to prevent the red window from advancing.

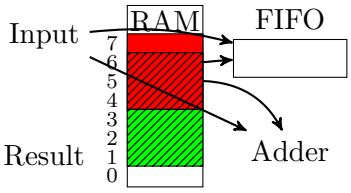
For the third case, the Intermediator would start storing values before they are completely accumulated if the fading window fades too quickly. However, all the values are accumulated after 80 cycles in the fading window.



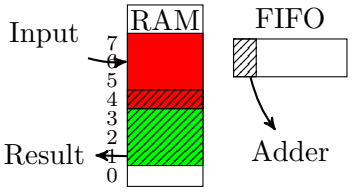
(a) First clock cycle, 1 pair of values get sent to the adder.



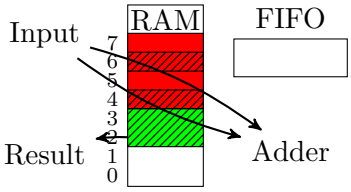
(b) Second clock cycle, 2 values gets stored in the RAM.



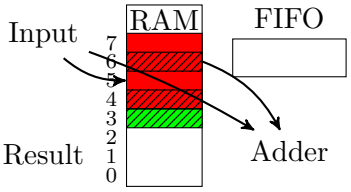
(c) Third clock cycle, the 2 inputs correspond to occupied cells in the RAM block.



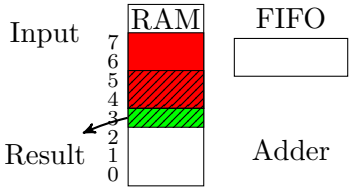
(d) Fourth clock cycle, 1 value gets stored in RAM.



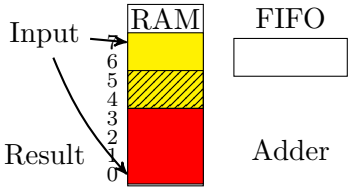
(e) Fifth clock cycle, the row indexes of the 2 inputs equal each other.



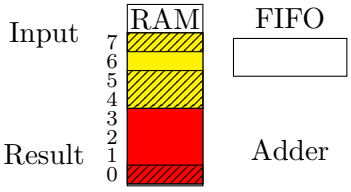
(f) Sixth clock cycle, 1 pair of values get sent to the adder, and 1 element gets stored in RAM.



(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 5.5: 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.

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.

5.2.4 Dual-port 1-bit Wide Memory

Since the Intermediator needs to know that occupied/vacant status of each slot in memory and update this status in a single clock cycle, we need a 1 bit RAM to keep track of this. 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 limited the number of active (red) intermediate values to 32. In our new design we use distributed RAM with a width of 1 bit to keep track of the state of each slot.

We implement a special case of the memory developed in **?**) to achieve this. This requires the use of distributed RAMs with only one read-only and one write-only port. 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 RAMs with one read port and one write port.

With a clever arrangement of 4 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

Table 5.5: MAC throughput

hello world

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 **??**.

5.3 Results

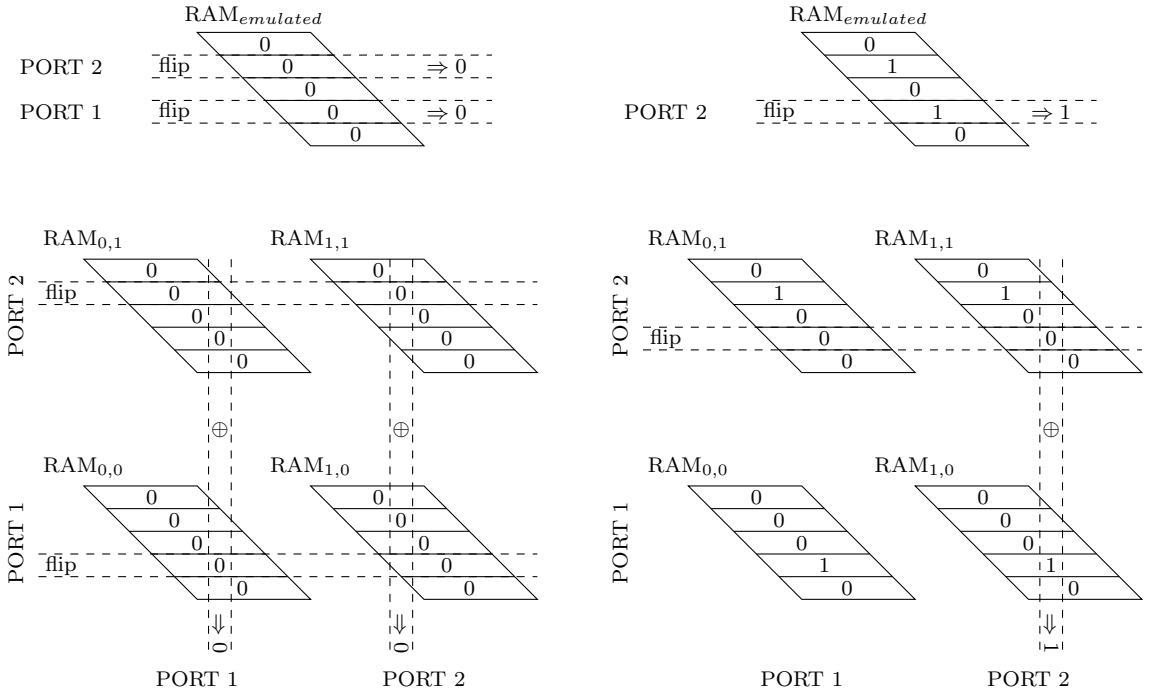
The two important metrics for most MACs are the throughput and synthesis results.

5.3.1 Throughput

Although the MAC rarely stalls in theory. We thought it would be important to show this in practice as well. We use the matricies from the benchmark and show the percent of time the MAC stalls in each case. We should this in figure X.

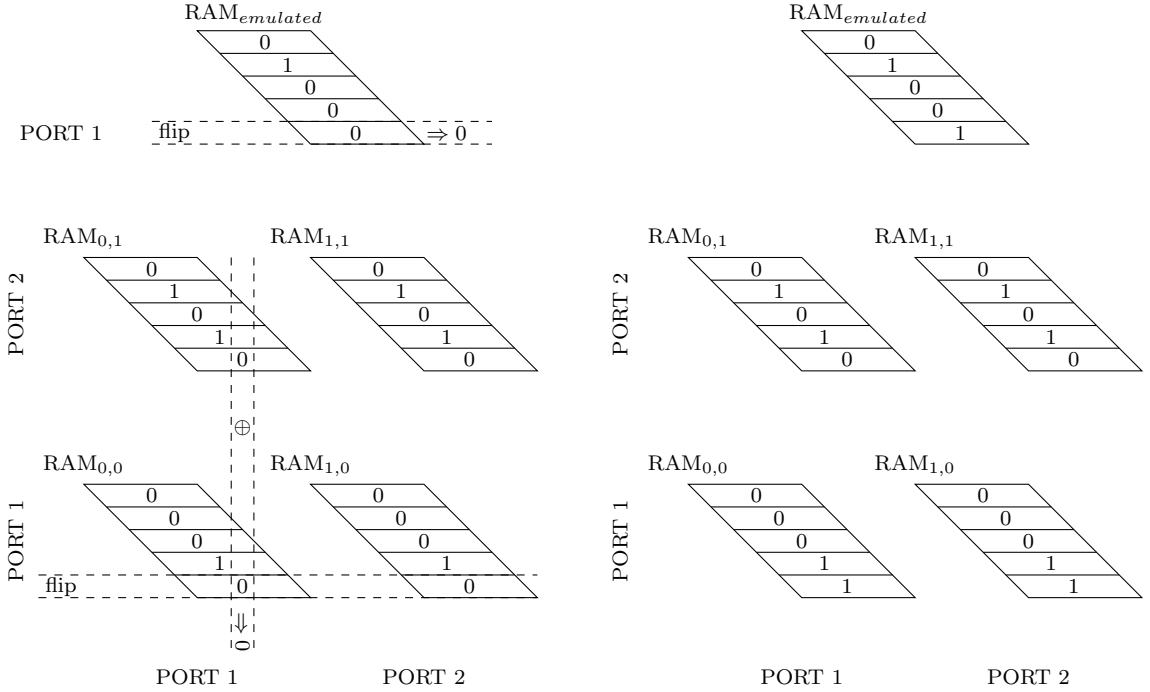
5.3.2 Synthesis

We placed and routed one MAC using xst (Xilinx Synthesis Tools). The frequency of the MAC was 263Mhz. The MAC used 3,105 Registers, 3,236 LUTs, 5.5 RAM blocks, and 7 DSP (multiplier) blocks. By default xst uses RAM blocks for the FIFO even though their depth is equal to 32. XST can be forced to use LUTs, which would use 112 more LUTs and 3.5 fewer RAM blocks.



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

CHAPTER 6. SPARSE MATRIX COMPRESSION

Although we address this pillar last we view it as the most important. We previously implemented matrix compression in **?**). However, we made the mistake of combining the two types of compression, index and floating point into one scheme. This simplified some aspects of the design. For example, this compression only needed to keep track of one data stream. Combining the two parts also makes sense if the two are correlated. In the extreme case the values in matrix could be calculated from the indices. However, we do not see an easy way to use this for general sparse matrix compression.

This chapter is dedicated to sparse pattern matrix compression, which we call SMC. In the next chapter we discuss our floating point compression, which we call fzip. This chapter starts with Section **??** discussing related work. Then, Section **??** discusses the analysis of delta compression. Then, Section **??** discusses our implementation. Then, Section **??** discusses the hardware decoder for SMC. Lastly, Section **??** discusses something.

6.1 Related Work

Some work exists on compressing indices beyond coordinate (COO) format. The most obvious one and the one mentioned back in Chapter **??** is compressed sparse row (CSR) format. The majority of work on computing SpMV on FPGAs uses CSR [**?**)]. However we did find one alternative that uses a format called CVBV [**?**)]. Also, **?**) uses index compression to speed up their CPU implementation. In addition to these we also look at how well gzip can compress indices.

Since both CVBV and the CPU method benefit from our RCR traversal we use that when analyzing them rather than row major traversal.

6.1.1 CVBV

Compressed Variable Bit Vector (CVBV) is a format created by [**?**)]. This implementation has two streams, which we will call the code stream and the argument stream. The code stream stores one 4 bit code for each delta. The first bit indicates what the type of the code is, dense (1) or regular (0). If the bit equals 1 (a dense code type) then the delta equals 1 and the value in the argument indicates how many deltas equal to 1 follow. If the bit equals 0 (a regular code type) then the delta equals the value in the argument.

The other 3 bits indicate how many nibbles are in the argument. The argument is taken from the argument stream. The results of this implementation is in Table **??** column 4.

6.1.2 CPU method

The method in [**?**)] is similar but targeted for CPUs rather than FPGAs. Again there are 2 streams a code stream and an argument stream. The codes are 2 bytes long. Instead of one code per delta there is one code per array of deltas. The first byte indicates what size the deltas are, 1, 2, 4, or 8 bytes, and in the array is the last one in the row. The second byte indicates the length of the array.

The argument stream can be padded to make sure the data types align properly. The whole propose of this implementation is that natïve data types are used so the CPU can process them faster. The compression results of this implementation is in Table **??** column 5.

6.1.3 gzip

We also choose to use the general compression program gzip in the comparison as well. Table **??** shows the compression of gzip on top of the CSR format. gzip does very well, partly because column indices often repeat.

6.2 Analysis

We did some analysis on the distribution of deltas to come up with our implementation. The most important analysis was the distribution of delta lengths Table **??**. This table show

Table 6.1: This table shows the number of bytes per non-zero value the given index compression scheme achieves. (No floating point values are being compressed here.)

Matrix	**COO**	**CSR**	**CSR.gz**	**cvbv**	**cpu**	**smc**	**smc.gz**
cant	8.00	4.06	0.40	0.51	1.01	0.26	0.04
consph	8.00	4.06	0.19	0.45	1.01	0.26	0.03
cop20k_A	8.00	4.18	1.07	0.99	1.01	0.64	0.54
dense2	8.00	4.00	0.03	0.00	1.01	0.13	0.00
mac_econ_fwd500	8.00	4.65	1.48	1.32	1.01	0.91	0.48
mc2depi	8.00	5.00	1.78	1.12	1.01	0.41	0.02
pdb1HYS	8.00	4.03	0.14	0.24	1.01	0.20	0.07
pwtk	8.00	4.07	0.16	0.23	1.01	0.19	0.01
qcd5_4	8.00	4.10	0.31	0.62	1.01	0.28	0.01
rail4284	8.00	4.00	1.41	0.62	1.01	0.50	0.45
rma10	8.00	4.08	0.20	0.38	1.01	0.24	0.09
scircuit	8.00	4.71	1.61	1.11	1.01	0.76	0.61
shipsec1	8.00	4.07	0.20	0.45	1.01	0.22	0.06
webbase-1M	8.00	5.29	1.35	1.09	1.01	0.58	0.31
average[a]	8.00	4.33	0.79	0.70	1.01	0.42	0.21

[a] Excludes the dense matrix.

the distribution of deltas for the RCR traversal. In this case we set the subwidth to 8 and the subheight to 512. There are some key characteristics to notice. First, about half of the deltas equal 1. Second, on average less than 5% of the deltas are more than 512. Third, with some eye squinting the frequency distribution of the deltas is approximately a decreasing exponential.

6.3 Sparse Pattern Matrix Compression (SMC)

We approach the problem of how to encode the series of deltas by giving each delta a code. We also created a special new line code. However, there are too many delta lengths to give each one a unique code. So we have 3 types of codes:

1. Constant offset code.
2. Variable offset code.
3. New line code.

The constant offset codes represent small deltas and decode as the exact value of the delta. The variable offset codes represent larger deltas and decode as the bit length of the delta ($\log_2(\delta)$). The new line codes represent the end of the 512 row section. After a newline code is read the running 'major' row index will increment and the running 'minor' row index will reset to -1 and all of the running column index will reset to -1.

Based on the analysis in Table **??** we choose to have 32 constant offset codes. Using the deltas we can determine the frequencies of each code. With these frequencies we can create the codes using Huffman encoding [**?**)].

As an example, consider the example matrix back in Chapter **??** in Equation **??**. Using row-major traversal (rather than RCR traversal) the deltas are:
1, 3, 3, \n, 5, 3, \n, 2, 1, 3, 1, \n, 1, 4, \n, 3, 1, 3, 1, \n, 2, 3, \n, 2, 1, 3, 2, \n, 3, 1, 1, 1

Using 2 (rather than 32) constant offsets, the codes and their frequencies are:

1. 1 : 10
2. 2 : 4

Table 6.2: The distribution of the bit lengths required to store the delta length when using RCR traversal with the subheight set to 512 and the subwidth set to 8.

Matrix	**1**	**2**	**3-4**	**5-8**	**9-16**	**17-32**	**33-64**	**65-128**	**129-256**	**257-512**	**512+**
cant	72.2%	6.3%	6.8%	12.3%	0.7%	0.0%	0.0%	1.0%	0.0%	0.0%	0.6%
consph	76.6%	3.1%	5.9%	11.6%	0.8%	0.0%	0.0%	0.0%	0.1%	1.0%	0.8%
cop20k_A	51.1%	1.9%	3.2%	22.9%	2.4%	1.0%	1.0%	1.0%	0.8%	0.7%	14.0%
dense2	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
mac_econ_fwd500	8.2%	7.8%	7.3%	19.0%	12.0%	15.2%	6.3%	5.1%	2.8%	7.1%	9.2%
mc2depi	21.8%	0.0%	0.0%	24.9%	43.8%	0.0%	0.0%	0.0%	0.0%	0.1%	9.4%
pdb1HYS	88.0%	1.1%	3.6%	5.9%	0.0%	0.3%	0.2%	0.2%	0.1%	0.1%	0.4%
pwtk	87.7%	2.7%	3.0%	5.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.8%
qcd5_4	66.0%	0.0%	8.2%	19.7%	3.6%	0.0%	0.0%	0.2%	0.0%	0.2%	2.1%
rail4284	71.2%	1.0%	1.5%	5.3%	0.4%	1.0%	1.5%	1.5%	1.6%	2.0%	12.9%
rma10	80.6%	1.4%	6.5%	8.8%	0.2%	0.8%	0.3%	0.1%	0.1%	0.2%	0.9%
scircuit	35.3%	2.8%	3.2%	25.7%	8.0%	3.5%	2.7%	2.3%	1.6%	1.1%	13.7%
shipsec1	78.4%	0.0%	6.1%	12.2%	1.1%	0.0%	0.3%	0.2%	0.3%	0.2%	1.3%
webbase-1M	14.6%	3.5%	2.3%	37.7%	28.7%	1.0%	0.8%	0.5%	0.4%	0.4%	10.1%
average[a]	57.8%	2.4%	4.4%	16.3%	7.8%	1.8%	1.0%	0.9%	0.6%	1.0%	5.9%

[a] Excludes dense matrix

3. 3-4 : 9

4. 5-8 : 1

5. \n : 7

This results in 5 codes: 2 constant offset codes, 2 variable offsets codes, and 1 newline code. Now these codes are given variable length codes through Huffman encoding.

Huffman encoding first creates a Huffman tree, which then is used to create the codes. The tree is created as follows. First, create a lone node for each code. Second, sort the nodes by frequency. Third, pop the two nodes with the lowest frequency from the list, and make the two nodes children of a new node. Fourth, set the frequency value of this new node to the sum of the frequencies of the children. Fifth, insert the new node back into the sorted list. Then repeat the third through fifth steps until the list only contains one node. Figure **??** shows the creation of the Huffman tree. The following is the sorted list at each iteration:

1. (1 : 10), (3-4 : 9), (\n : 7), (2 : 4), (5-8 : 1)

2. (1 : 10), (3-4 : 9), (\n : 7), (a : 5)

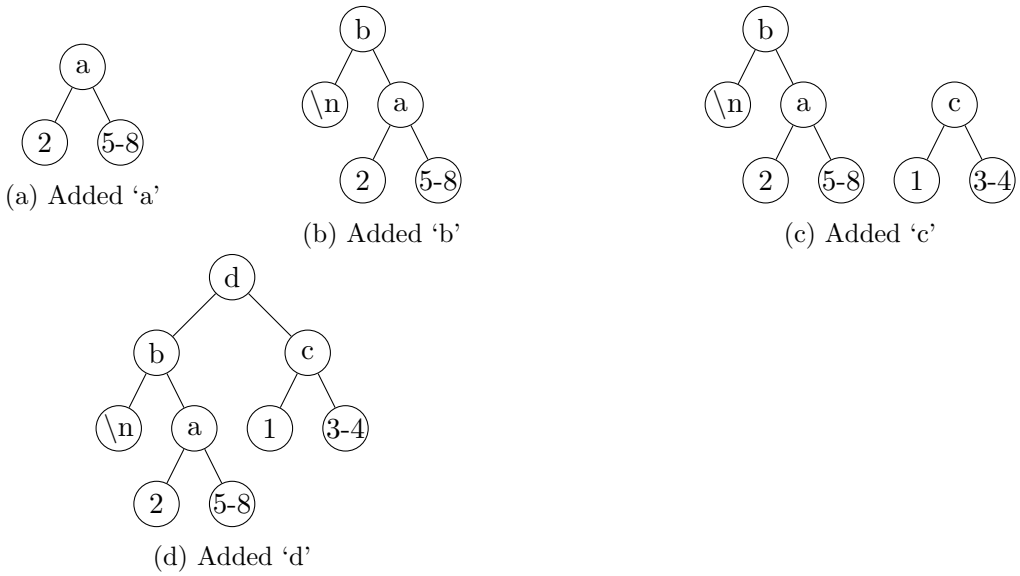
3. (b : 12), (1 : 10), (3-4 : 9)

4. (c : 19), (b : 12)

5. (d : 31)

To encode a large delta with a variable length code, the delta (minus the most significant bit) is pushed onto an argument stream. In the end the SMC file has three parts: the header, the dictionary, the codes stream, and the argument stream.

To make decoding easier we set the maximum code length to 9. We do this by artificially increasing the frequency of each code to a minimum of $\frac{nnz}{2^9}$. In the SMC file a code dictionary needs to be present so that the file can be decompressed. For easier decoding we have 2^9 records with all the information to decode them.



(a) Added 'a'

(b) Added 'b'

(c) Added 'c'

(d) Added 'd'

Figure 6.1: Each step of creating the Huffman tree.

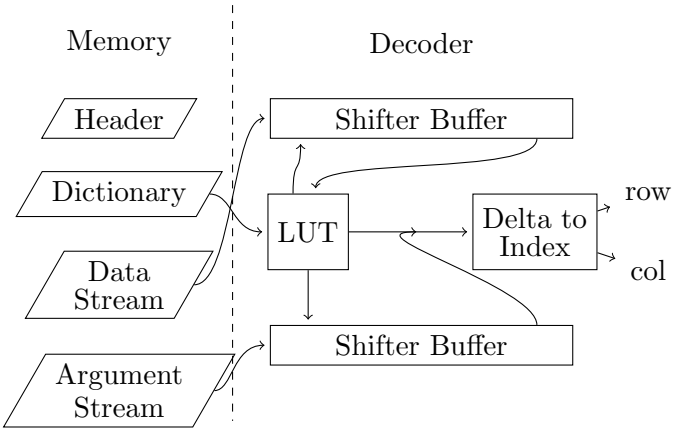


Figure 6.2: The hardware design of the SMC decoder.

6.4 The SMC decoder

The hardware decoder relies on lookup tables to decode the two streams (see Figure **??**). First the dictionary is loaded onto a large 512 value lookup table (LUT). This lookup table uses the code length to determine how much to right shift the first shift buffer. The LUT then determines if the second shift buffer is needed (the code is a variable length delta). In that case, the second shift buffer is right shifted and the output is used to create the delta. Then the delta (or newline) is sent to the running row and column index and the current row and column index is outputted from the decoder.

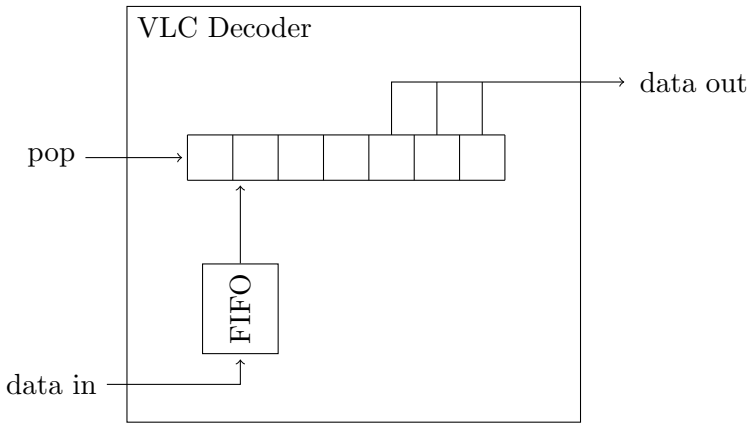
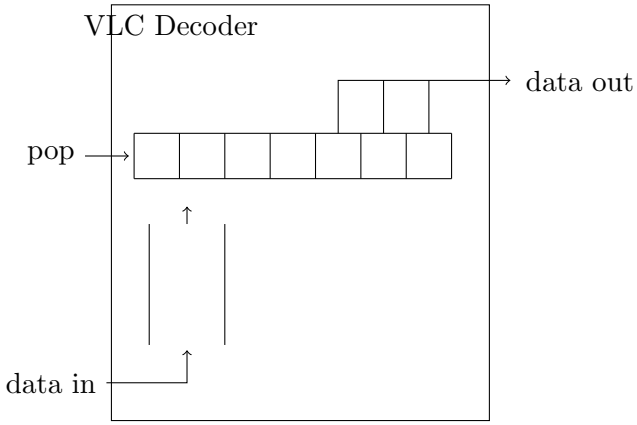


Figure 6.3: VLC Decoder.



(a) VLC Decoder

We do not have the exact area and performance of this decoder because we only designed a decoder that combined the index and floating point decoding. However, the first shift buffer uses 293 LUTs and 60 registers. The second shift buffer uses 469 LUTs and 82 registers. The combined decoder has a top frequency of 150 Mhz after place and route using xst.

6.4.1 VLC Decoder

6.4.2 Huffman Decoder

6.5 Results

Table **??** shows the results of SMC compression. As seen, it outperforms the previous implementations. In addition, we looked at compressing the smc file further with gzip. This achieve a good deal of addition compression. One reason for this additional compression is that

it compresses the repeating deltas equal to 1 in the dense sections of the matrix.

CHAPTER 7. FLOATING POINT COMPRESSION

Floating point compression is the second half of matrix compression. Figure **??** shows a comparison of compression schemes. In the end, we created a program and library called fzip. In total, fzip takes advantage of 2 compressible features of datasets: 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. So taking advantage of sequences that are more than 8 bytes long is difficult. In the remainder of this chapter we talk about an analysis of floating point datasets (Section **??**), our approach to floating point compression (Section **??**), our hardware decoder (Section **??**) and our results (Section **??**).

7.1 Related Work

It was noted in **??**) that sparse matrices often have repeated values. This is the focus of our value compression. Our previous work (R^3) had a simple scheme using this feature. 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 **??**.

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 good compression scheme should take less than 8 bytes per element. We looked at **?**) describing its own compression scheme, FPC. This scheme looks for repeated patterns. However it does not exploit the fact most of its compression comes from exact (8 byte) value repeats. To illustrate this point we created an "anti-FPC" dataset (Figure **??**).

gzip performs quite well too. We have a general understanding of how gzip works. We

Table 7.1: Detailed value compression analysis and performance comparison, in terms of bytes per non-zero value.

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

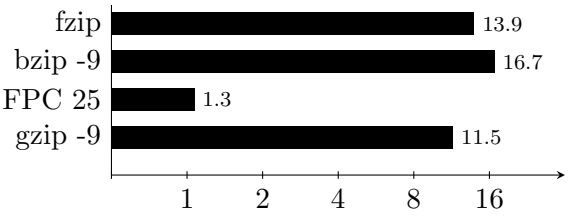


Figure 7.1: We engineering a dataset to make the performance of FPC look bad compared to other programs. Although unfair, this shows a type of pattern that FPC does not exploit and other programs do. This problem exists because FPC only uses predictors for compression.

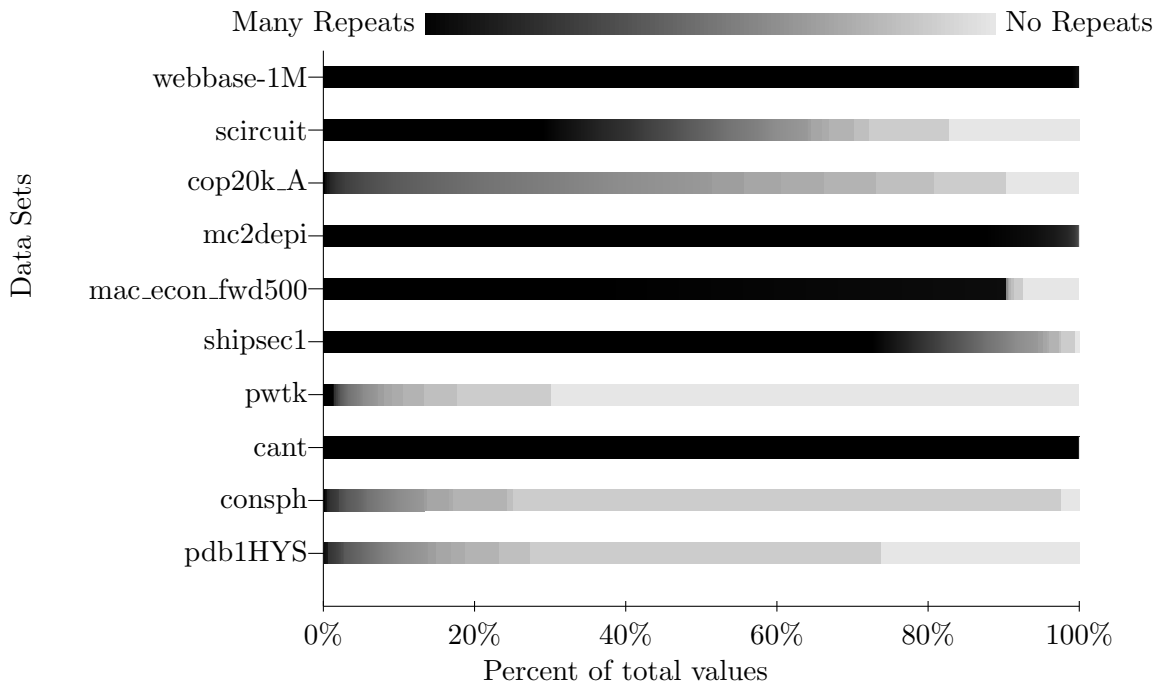


Figure 7.2: The above figure shows the distribution of repeats in each dataset. Each shade represents a different number of repeats. For instance: ■:> 512, ■:16, ■:2, ■:1(no repeats).

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.

7.2 Floating-Point Value Analysis

Continuing the analysis from the beginning of this chapter, Figure **??** 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,

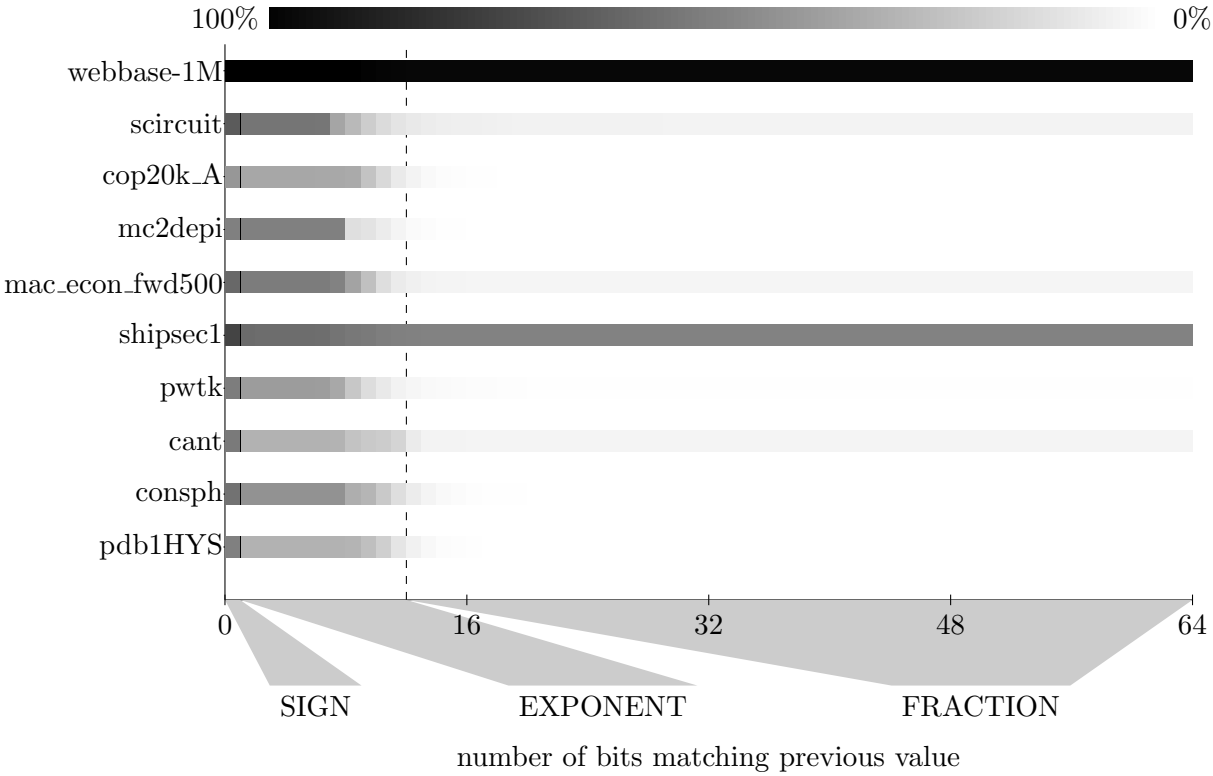


Figure 7.3: The above figure represents local prefix prediction. The figure shows the density function of 2 adjacent values sharing at least x number of prefix bits. All of the data sets start at $(0, 100\%)$. The curves end at the percent of values that are identical to their previous value for that dataset.

11 exponent bits and 52 fraction bits. Values close to each other in the dataset often share the same sign. (Some datasets only contain positive numbers.) Likewise, close values often share the most significant bits of the exponent. In fact, the bits in floating-point values already exist in most likely shared to least likely shared sorted order: {sign bit, most significant exponent bits, least significant exponent bits, most significant fraction bits, least significant fraction bits}.

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

Datasets might also have repeating patterns of values. For example, the sequence 1.0, 2.0, 3.0, 1.0, 2.0, 3.0 has an obvious pattern. One can use the Burrows Wheeler Transform**?**) to

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

analyze these patterns. Figure **??** describes this algorithm some, however, many other sources describe this algorithm in more detail, for example **??**). Figure **??** 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, but these are also the matrices with a lot of repeats to begin with.

7.3 Our Approach

Since BWT does not provide great compression, depends on the traversal of the matrix and is not hardware amenable, we designed fzip to only use prefix and repeat compression.

7.3.1 Prefix Compression

fzip uses arithmetic encoding followed by Huffman encoding to encode common prefixes. To begin with, fzip creates a large tree to represent all the values in the array. Figure **??** shows an example tree for a small dataset. The tree follows the following rules: each node has up to two children. Each edge represents a 1 bit or a 0 bit. Each node in the tree represents a prefix. The root node represents “” or no prefix. Each node also has a weight, which represents the

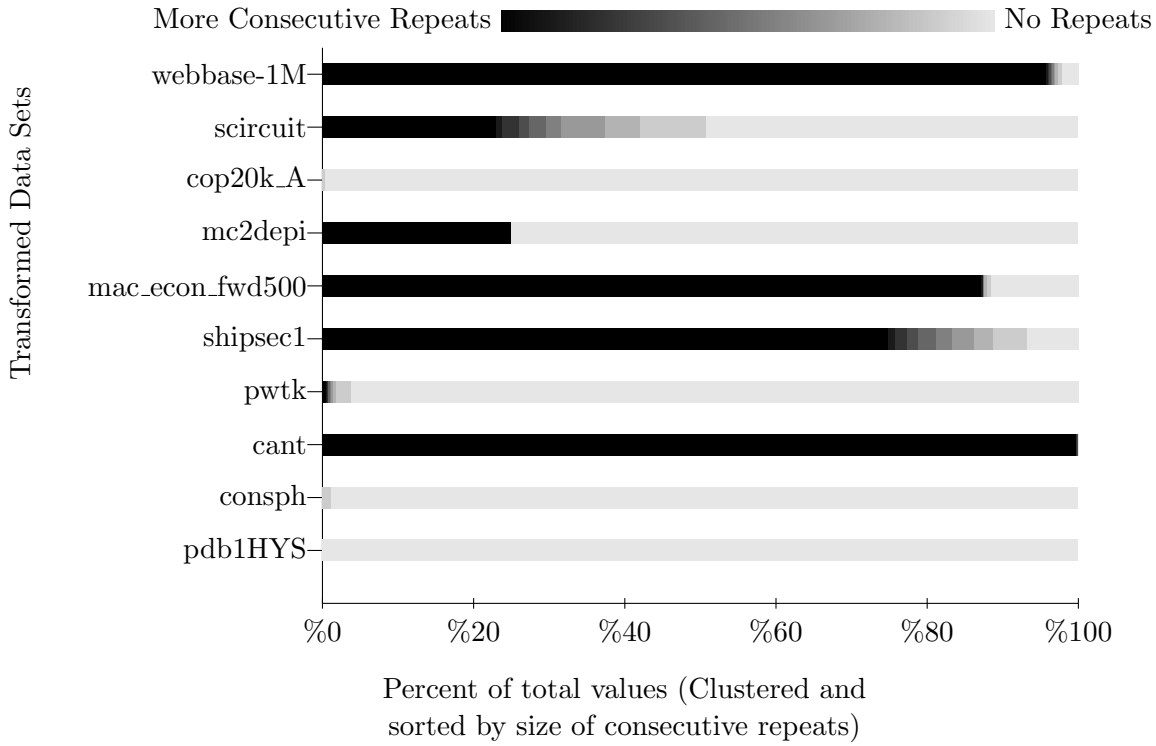
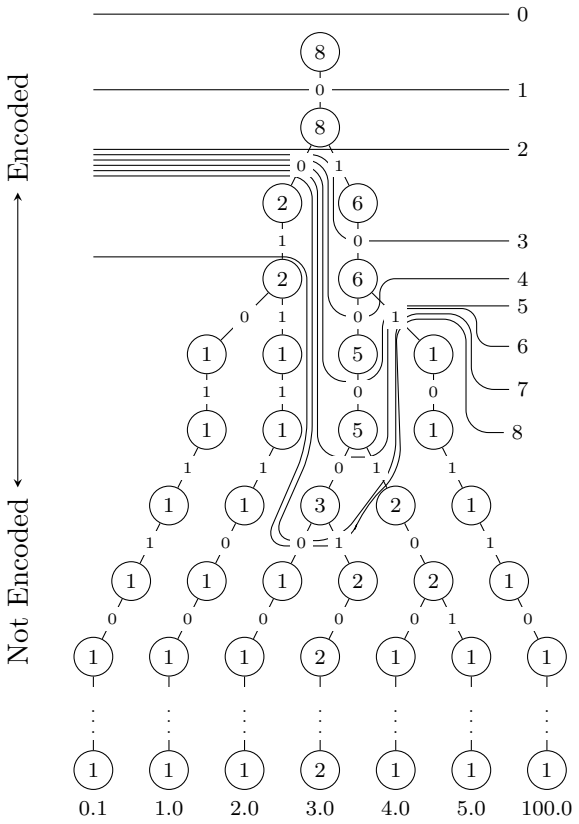
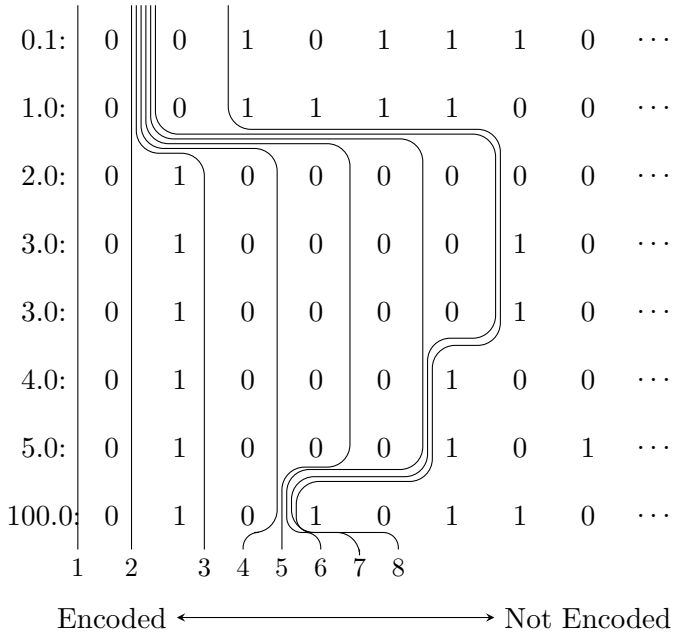


Figure 7.5: Pattern analysis using the Burrows-Wheeler Transform. Each shade represents the number of consecutive repeats in a repeating sequence. ■ represents sequences longer than 9. ■ represents sequences of length 5. ■ represents sequences equal to 1 (non-repeating).



(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 7.6: 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.

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. 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 **??**:

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 256 records.

7.3.2 Repeated Value Extension

Prefix compression does not compress all of the repeated values. So, fzip extends prefix compression to specifically include commonly repeated values. Again explaining why repeated

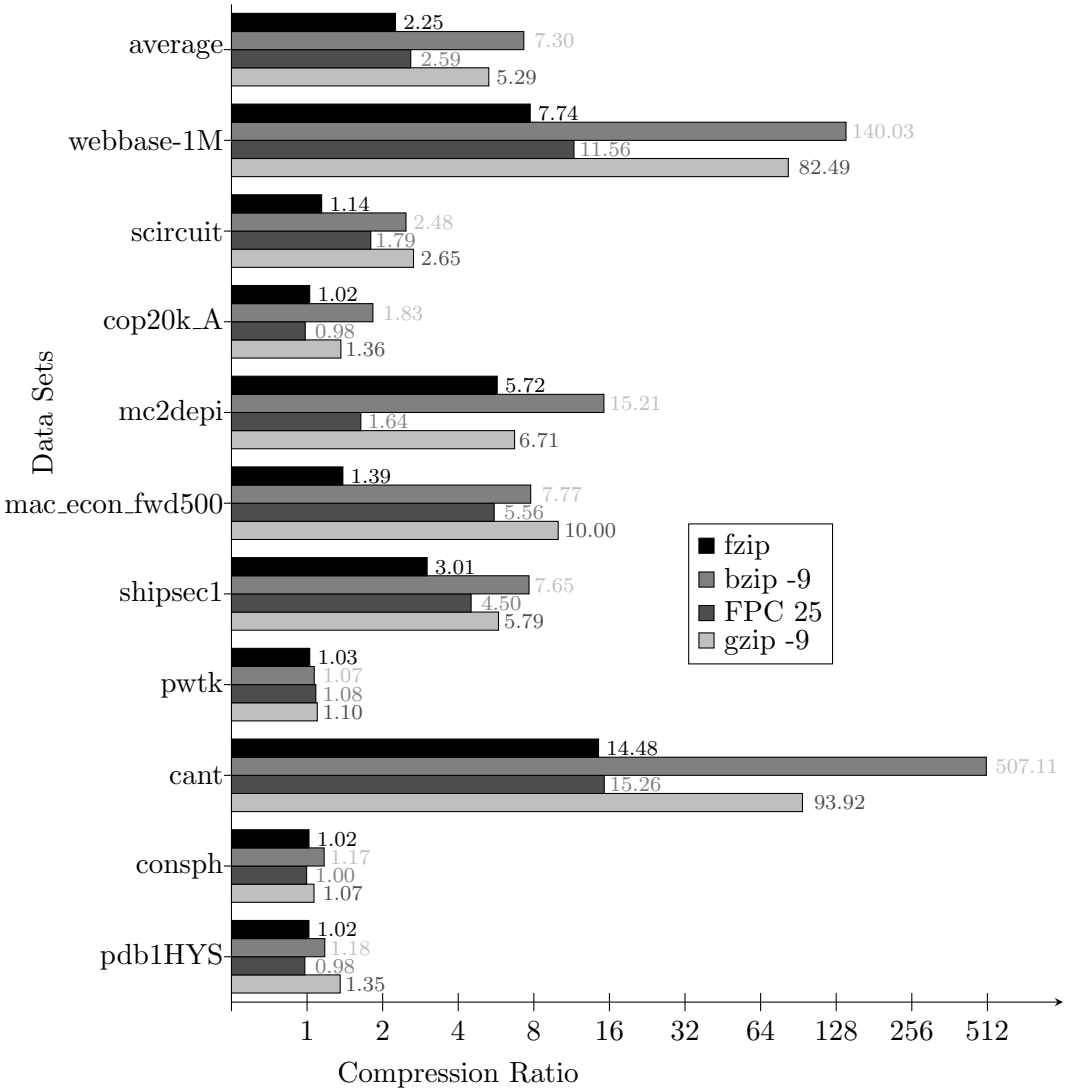


Figure 7.7: The comparison of different compression schemes shows fzip performs competitively.

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 add a special code to the set of prefix codes. We limit the number of repeats to 8192, so when this code is encountered 13 bits will be on the not-encoded stream indicating the address of the common value.

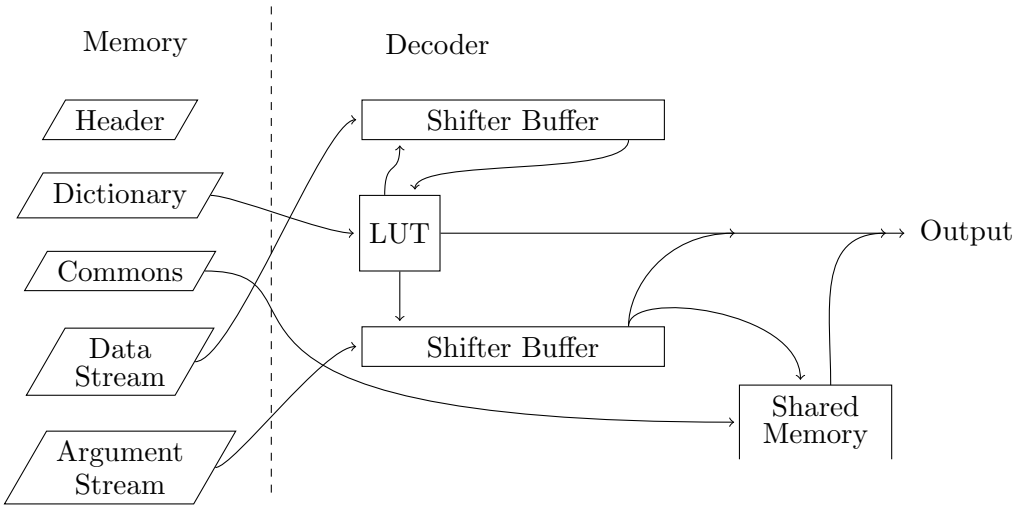


Figure 7.8: The hardware design of the fzip decoder.

7.4 The fzip decoder

Like the SMC decoder from Section **??** in the previous chapter, the fzip hardware decoder relies on lookup tables to decode the two streams (see Figure **??**). First the dictionary is loaded onto a large 512 value lookup table (LUT). This lookup table uses the code length to determine how much to shift the first shift buffer. The LUT then determines if the second shift buffer is needed (the code is a variable length delta). Then the delta (or newline) is sent to the running row and column index and the current row and column index is outputted from the decoder.

We do not have the exact area and performance of this decoder because we only designed a decoder that combined the index and floating point decoding. However, the first shift buffer uses 293 LUTs and 60 registers. The second shift buffer uses 957 LUTs and 146 registers. The combined decoder has a top frequency of 150 Mhz after place and route using xst.

7.5 Results

We present fzip's results in Figure **??**. As seen fzip does not perform quite as well as its competitors. However, fzip is very limited in what it uses for memory space when decompressing. Increasing the common values space from 64KB to an 'unlimited' space and increasing the number of prefix codes does increase the compression ratio, but this no longer makes it

possible to create a hardware decoder. As the next chapter shows creating an efficient 64KB shared memory on an FPGA is not a trivial task.

CHAPTER 8. 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.

8.1 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 **???**), cannot support our desired clock frequency. *Replication*, seen in **???**), and *Live Value Table*, seen in **???**), store excessive redundant information information in RAM blocks. This leaves *Banking*, seen in **???**), which can scale to 16 or more ports.

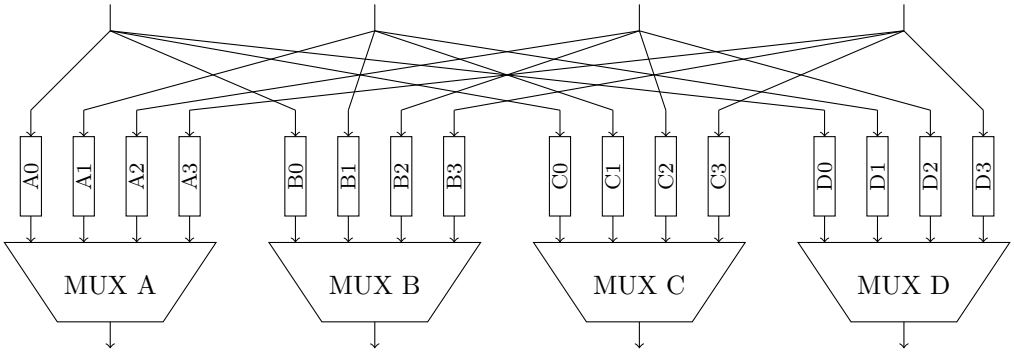


Figure 8.1: The fully-connected interconnect network.

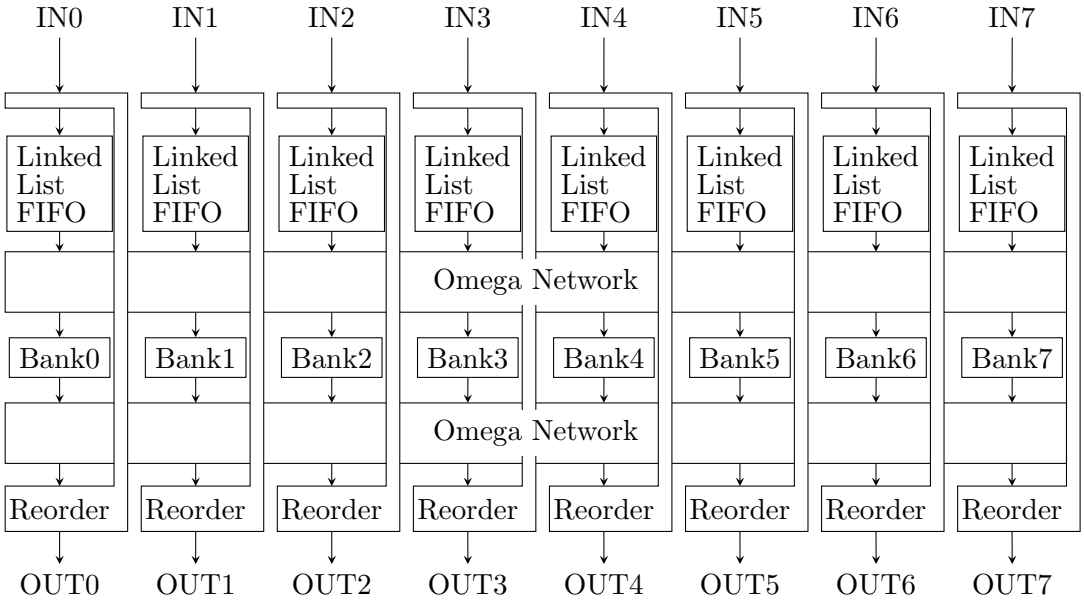


Figure 8.2: In the Omega multi-port memory all the buffering occurs in the linked list FIFOs. The use of multi-stage interconnect networks, in this case Omega networks, helps reduce the area of the design.

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

8.2 Omega Multi-port Memory

The Omega multi-port memory (Figure **??**) 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

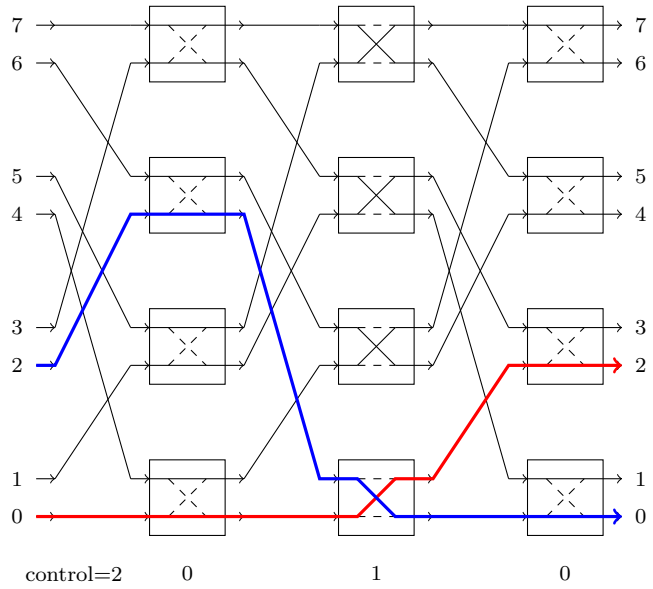


Figure 8.3: An 8-by-8 Omega network. We turn columns on or off to rotate between different routing configurations.

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.

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

8.2.2 Omega Network

An Omega network consists of columns of Banyan switches **??**). A Banyan switch synthesizes to two multiplexers. In the on state, the switch crosses data over to the opposite output

port. As an illustrative example, the second column in Figure **??** 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 **??** 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 **??**, the control bits equal 010_2 or 2 and input port 2 (010_2) routes to output port 0. Not coincidentally, the same configuration routes in reverse. Input port 0 routes to output port 2 and input port 2 routes to output port 0. This means the design can use identical control bits for both the receiving and sending Omega networks.

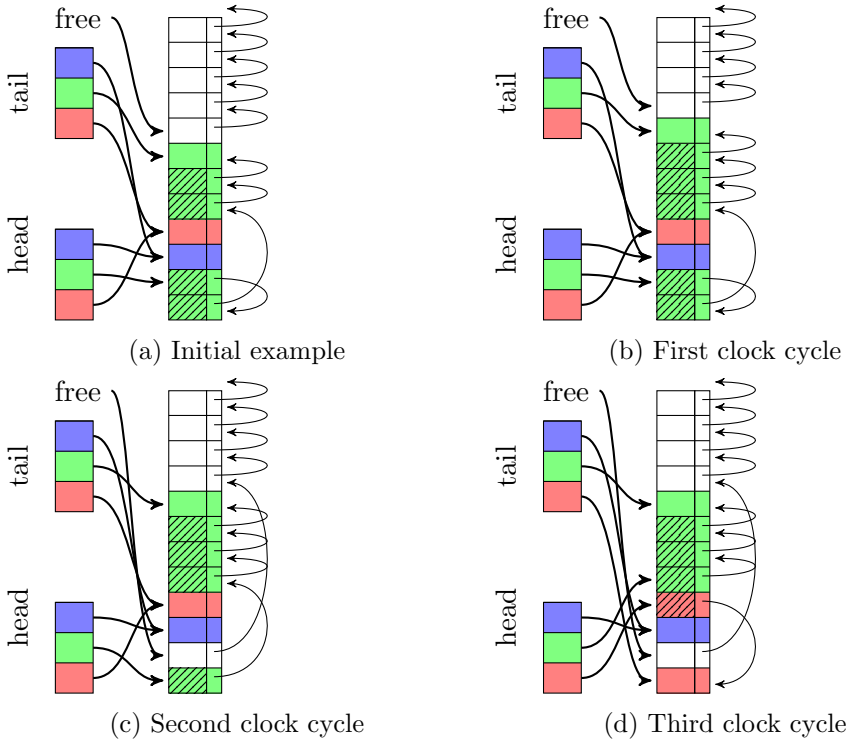
In the Omega multi-port memory design, the control for this network increments every clock cycle. As an example, input port 5 would connect to output port 5, then port 4, 7, 6, 1, etc., until it cycles around again. This means each input connects to each output an equal number of times.

8.2.3 The Linked List FIFO

The partnering hardware structure, the linked list FIFO (Figure **??**), 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 [**??**)].

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 **??**, which uses a 16 deep RAM and 4 FIFOs.

In the initial state (Figure **??**), 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 $TOTAL\_DEPTH - FIFO\_COUNT$.



(a) Initial example (b) First clock cycle

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

Figure 8.4: A linked list FIFO during 3 clock cycles of operation

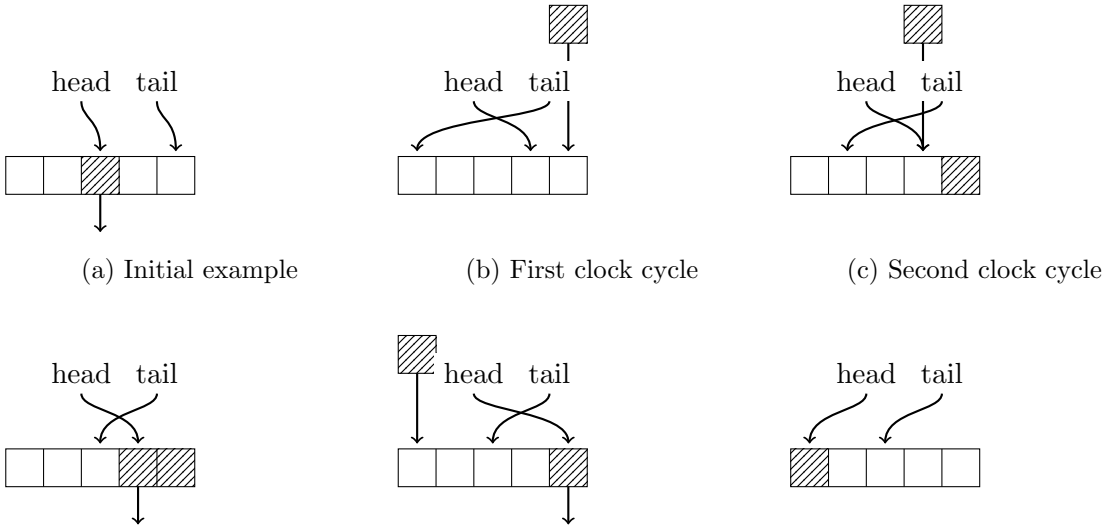
On the first clock cycle (Figure **??**), the linked list FIFO receives a push containing a red message. The new red message gets stored in the reserved space at the tail of the red linked list. The free linked list pops one space. That space gets pushed on to the red linked list.

On the second clock cycle (Figure **??**), 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 **??**), 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.

8.2.4 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,



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

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

Figure 8.5: Reorder queue example.

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

On the first clock cycle (Figure **??**), 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 **??**), 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 **??**), a message arrives at the head of the reorder queue.

On the fourth clock cycle (Figure **??**), 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 **??**), 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.

8.3 Evaluation

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

8.4 Results

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

8.4.1 Varying the number of ports

In terms of area, Figure **??** 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 **??** shows that increasing the number of ports decreases throughput, and Table **??** shows increasing the number of ports increases latency. As expected,

Table 8.1: 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.

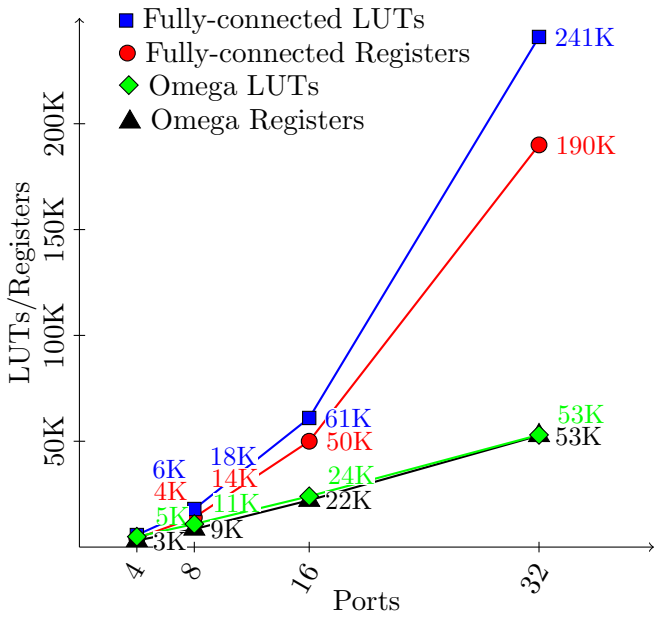


Figure 8.6: The effect of varying the number of ports on FPGA resource utilization (area). The Fully-connected memory grows by approximately N^2 and the Omega memory grows almost linearly.

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.

8.4.2 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 **??** 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 buffer depth.

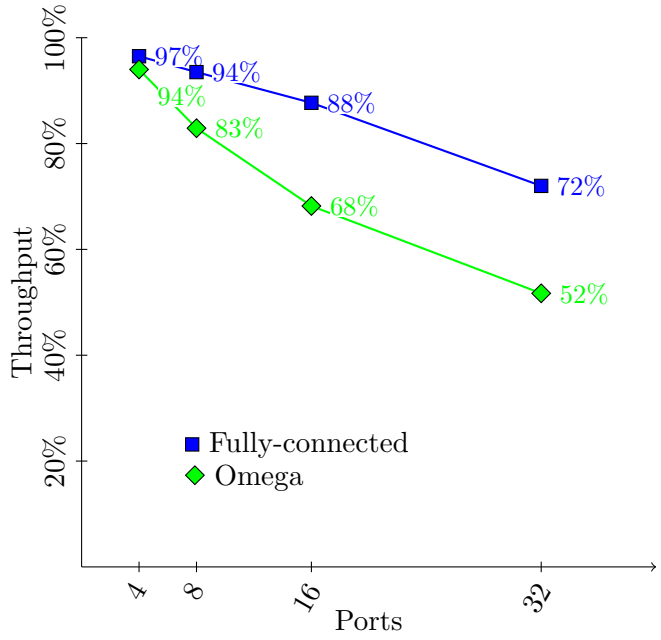


Figure 8.7: The effect of varying the number of ports on throughput of the random memory access benchmark on the small resource memories.

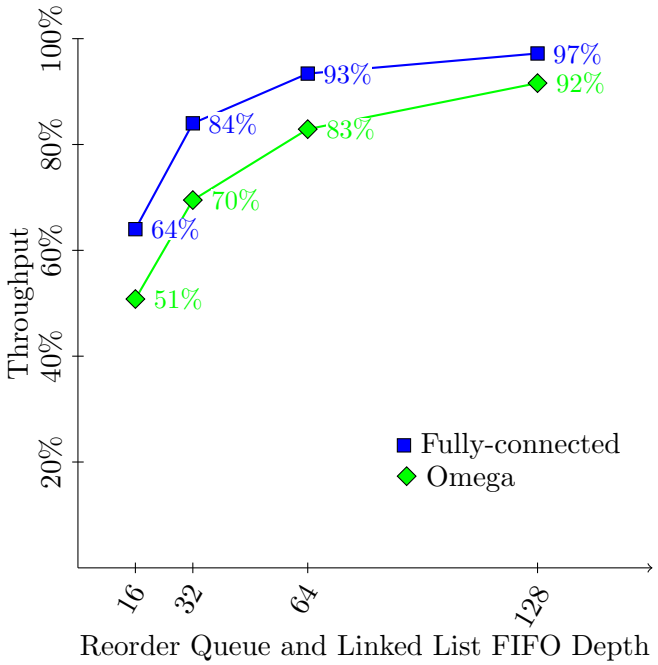


Figure 8.8: The effect of varying the depth of the linked list FIFOs and reorder queues on throughput of the random memory access benchmark (using 8-port memories).

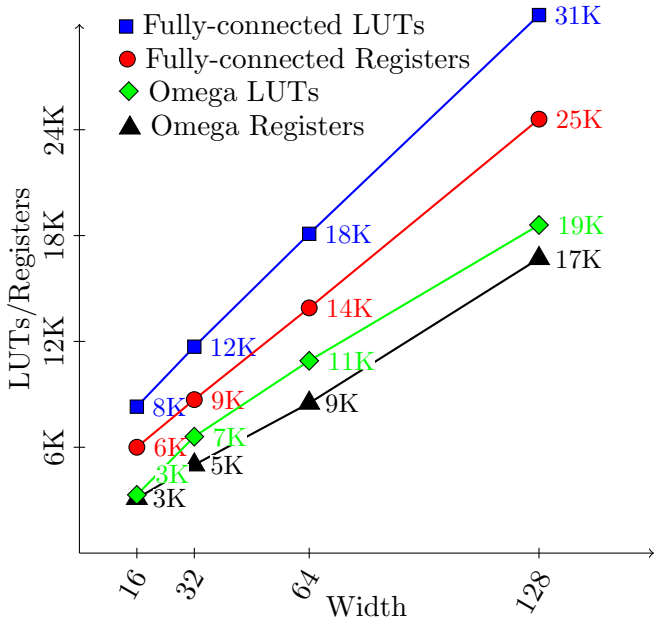


Figure 8.9: The effect of varying the bit-width of the memory on FPGA resource utilization.

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.

8.4.3 Varying the data bit width

Data width only effects resource utilization. As Figure **??** 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 9. HIGH LEVEL DESIGN

This chapter is about creating an implementation using the subcomponents mentioned in the previous chapter. The rest of the chapter is organized as follows. In Section **??**, We will look at the system design (everything higher than a single processing element). In Section **??**, we will look at our processing element design. In Section **??**, we will look at the file format. In Section **??**, we will look at how to deal with memory latency. Finally, in Section **??**, we look at the results achieve from the implementation.

9.1 System Design

Because each processing element acts independently during the SpMV computation it is fairly easy to come up with a high level organization. Systolic arrays work well on FPGAs **?**). So we will use a simple 1-D systolic array for communication (See figure **??**). The high amount of computation to instructions means the $O(p)$ time to send instructions is negligible. In addition, each PE connects to the shared memory and external memory. This platform has 16 virtual ports (8, 300Mhz ports multiplexed into 16 150Mhz virtual ports).

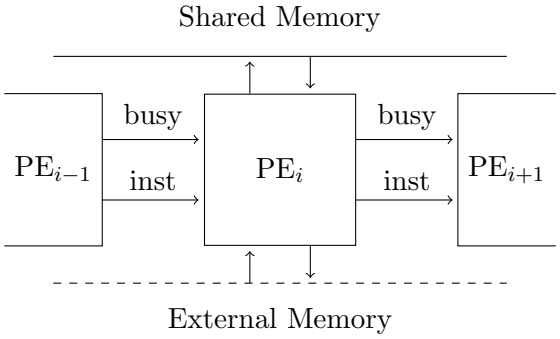


Figure 9.1: The processing element are arranged in a simple 1-D systolic array.

Table 9.1: The opcodes for the SpMV processor.

opcode	name	arg1	arg2	description
0000	NOP	N/A	N/A	This is the default signal sent to the PEs.
0001	RESET	N/A	N/A	This reset eliminates the need for a seperate reset signal to be routed to all the PEs.
0010	WRITE	register address	data	This loads data into the specified register.
0011	LOAD_DELTA_CODES	N/A	N/A	This loads the delta codes into the cooresponding lookup table in the decoder.
0100	LOAD_PREFIX_CODES	N/A	N/A	This loads the prefix codes into the cooresponding lookup table in the decoder.
0101	LOAD_COMMON_VALUES	N/A	N/A	This loads the 8,192 most common values into the shared memory.
0110	EXECUTE_SPMV	N/A	N/A	This instructs the PE(s) to perform the SpMV operation.
0111	READ	register address	N/A	This instructs the PE to return the data on the specified register.
1000	RETURN	register address	data	This is sent from the PE after the READ instruction is received.
1001-1111	RESERVED	N/A	N/A	N/A

9.2 Processor Design

One way to describe our processing element is that it is a very small instruction set processor for SpMV. However, the instructions are mostly for organization of the hardware not execution like in other work [**?**)]. The instructions are 64 bits wide and consist of 4 parts: a 4-bit opcode, a 5-bit PE ID, a 4-bit first argument, and a 51-bit second argument. The opcodes described in Table **??**.

There are 14, 48-bit registers on each processing element. The function of each register is described in Table **??**. All of these registers serve a purpose during the main stage (SPMV_EXECUTE). Registers 4, 5, 8 and 9 also have a function in the other stages (LOAD_DELTA_CODES,

Table 9.2: The registers in each SpMV processing element.

Register	Name	Description
0	y vector address	The address to store the next y value.
1	ending y vector address	The one over last address of the last y value.
2	x vector address	The static address of the x vector.
3	nnz MAC count down	Starts at nnz - 1 and decreases by one for each value that enter the MAC.
4	index opcodes address	The address of the next 8 byte chunk to request from the index opcodes stream. In the LOAD_DELTA_CODES, LOAD_PREFIX_CODES, and LOAD_COMMON_CODES operations this register stores the address of the next 8-byte chuck to read in.
5	index argument address	The address of the next 8 byte chunk to request from the index argument stream. In the LOAD_DELTA_CODES, LOAD_PREFIX_CODES, and LOAD_COMMON_CODES operations this register stores the address of where the next element is stored in on-chip memory.
6	fzip opcodes address	The address of the next 8 byte chunk to request from the fzip opcodes stream.
7	fzip argument address	The address of the next 8 byte chunk to request from the fzip argument streams.
8	ending index opcodes address	The one over last address of the space allocated to the index opcodes stream. In the LOAD_DELTA_CODES, LOAD_PREFIX_CODES, and LOAD_COMMON_CODES operations this register stores the one over last address of the data being read in.
9	ending index argument address	The one over last address of the space allocated to the index arguments stream. In the LOAD_DELTA_CODES, LOAD_PREFIX_CODES, and LOAD_COMMON_CODES operations this register stores the one over last address of the on-chip memory.
10	ending fzip opcodes address	The one over last address of the space allocated to the fzip opcodes stream.
11	ending fzip arguments address	The one over last address of the space allocated to the fzip arguments stream.
12	nnz index count down	Starts at nnz - 1 and decreases by one for each index pair that exits the decoder.
13	nnz fzip count down	Starts at nnz - 1 and decreases by one for each floating point value that exits the decoder.
14-15	reserved	Used for debugging.

LOAD_PREFIX_CODES, LOAD_PREFIX_CODES, and LOAD_COMMON_VALUES).

The debug registers give a better understanding of the operation of the processing element during the SpMV operation.

9.3 File format

We looked at previous file formats like gzip [**?**)] to help create a new file format. We start with a 256-byte header. This header has reserved values for future versions, pointers to the sections of the file, and other values about the matrix or compression streams. Each value in the header is 8-bytes. The specification of each value is described in Table **??**.

Three constant length sections after the header. First, the codes for the sparse pattern matrix compression. Second, the codes for the fzip compression. Third, the common floating point values (also for fzip compression). These sections will take a small fraction of the total file space for large matrices. For this reason we choose not to try to compress these sections further and just leave them as simple as possible.

Four data streams follow after all the decoding information, the code and argument stream for the indices and the code and argument streams for the floating point values. These streams have a variable bit length but there exists extra buffer bits to get to a 8 byte boundary. To make the design of the decoders easier there is at least 8 bytes of buffer space. For the fzip argument stream, there is at least 48 bytes of buffer space to increase throughput of decoding floating point values.

9.4 Memory Latency

Dealing with memory latency is not trivial. The difficulty lies in the fact we are reading 4 streams at different rates. Our platform has a 700 clock cycle external memory latency. We created a general approach with one tweak to achieve high throughput under these conditions.

The hardware consists of 3 parts. First, 4 FIFOs buffer the memory responses of the 4 different streams. Second, an 'in-flight-tracker' keeps track of the number of requests to ensure the FIFOs never overflow or have to stall the memory. Third, the request logic determines

Table 9.3: The header for our sparse matrix compression file format (SMC).

Number	Field	Description
0	RESERVED	
1	width	The width of the matrix.
2	height	The height of the matrix.
3	nnz	Number of non-zero values in the matrix.
4	spmCodeSTreamBitLength	Number of bits in the sparse pattern code stream.
5	spmArgumentStreamBitLength	Number of bit in the sparse pattern argument stream.
6	fzipCodeStreamBitLength	Number of bits in the fzip (floating point) code stream.
7	fzipArgumentStreamBitLength	Number of bits in the fzip (floating point) argument stream.
8-15	RESERVED	
16	spmCodesPtr	Offset from the start of the file to the start of the delta codes (will always be 256).
17	fzipCodesPtr	offset from the start of the file to the start of the fzip codes (will always be 4352).
18	commonDoublesPtr	Offset from the start of the file to the start of the 8,192 common floating point values (will always be 12544).
19	spmCodeStreamPtr	Offset from the start of the file to the start of the sparse pattern matrix code stream.
20	spmArgumentStreamPtr	Offset from the start of the file to the start of the sparse pattern matrix argument stream.
21	fzipCodeStreamPtr	Offset from the start of the file to the start of the fzip code stream.
22	fzipArgumentStream Ptr	Offset from the start of the file to the start of the fzip argument stream.
23	size	The total number of bytes in the file.
24-31	RESERVED	

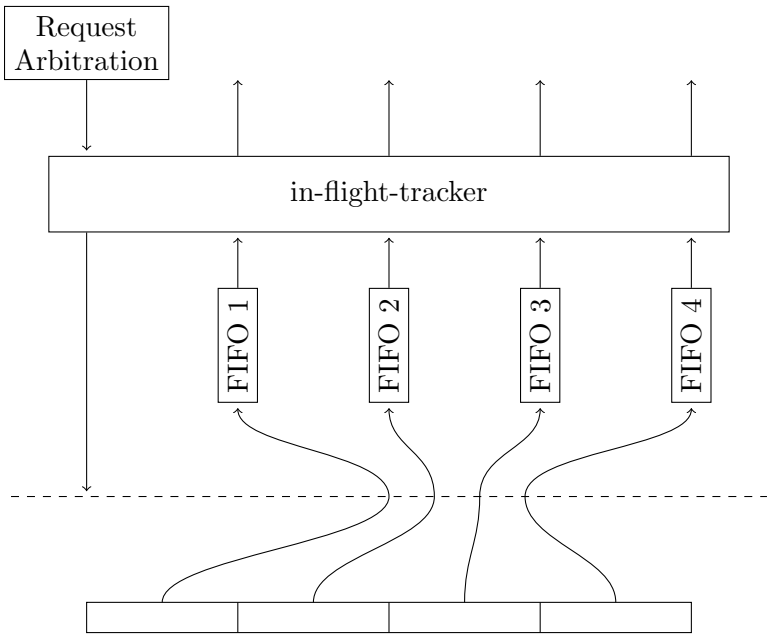


Figure 9.2: Memory request arbitration.

which (if any) stream should be read from on the current clock cycle.

We use a round robin for request arbitration. We use 3 different signals to determine when to move to the next stream: the tracker information, FIFO saturation and time. First, if the tracker indicates that the limit of in-flight requests has been reached the request arbiter switches to the next stream. Second, if the FIFO corresponding to the to the current stream is half full (indicating that the stream does not need to be read from so often) then the request arbiter switches to the next stream. Third, a timer insures each well get at most 32 requests before the request arbiter switches to the next stream.

In practice this still does not achieve good throughput when computing memory bound (difficult to compress) matrices. The problem is that the fzip argument stream (the non-encoded bits or common value index stream) ends up starving and slows down the decoder. To solve this we give priority to this stream by only moving from it when the tracker indicates no more in flight messages can be accepted. This means the timer and the FIFO saturation will not cause the request arbiter to stop reading this stream.

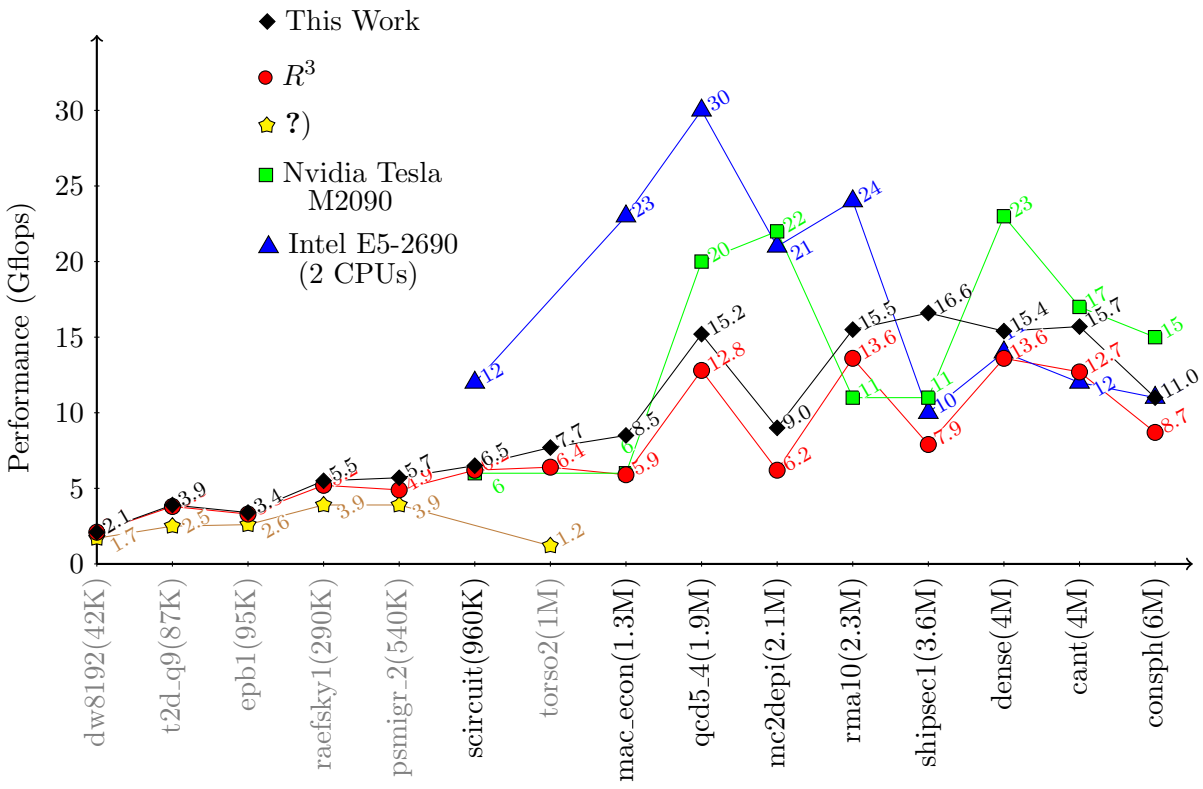


Figure 9.3: *nnz* vs Performance on each platform. As seen we achieve about a 20% increase in performance compared to R^3.

9.5 Results

R^3, our previous work, achieved up to 13.6 GFLOPS and an average performance of 7.6 GFLOPS. In this work we were able to achieve up to 16.6 GFLOPS and achieve an average performance of 9.4 GFLOPS.

Some trends do appear. Like in R^3 the new design does not achieve good performance on the smallest matrices. This is due to the memory latency of the platform. In fact, the smallest matrix (dw8192) is so small that all of the read requests occur before the first value is multiplied.

Another trend is that the more compressible matrices achieve the best performance. The 3 pattern matrices (qcd5_4, rma10, and dense) achieve the best performance.

In terms of area about half of the Xilinx Virtex-6 LX760 FPGA is utilized. The uses 255K out of 948K registers, 224K out of 474K LUTs, 515 out of 720 RAM blocks, 112 out of 864

DSPs (multiplier blocks).

CHAPTER 10. CONCLUSIONS

Achieving high performance SpMV on FPGAs requires optimizing several different problems. However we believe SpMV is an important computation and that FPGAs can outperform CPUs and GPUs in certain situations. 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.

10.1 Future Work

There are many different avenues for future work and places where this work can be used. The compression ideas presented in this dissertation can be ported to CPUs and may improve performance for very large matrices where external memory bandwidth is an issue (verses matrices that can fit on the 30MB caches CPUs now have). The Convey HC-2ex is dated and porting this code to another platform would allow this work to continue evolving.

Some of this dissertation work also applies to fields not related to SpMV. For example, the decompression hardware that we used for decompression floating point values was very similar to the decompression hardware used for indices. We suspect this may apply to other datasets as well. Having common decompression hardware available on modern CPUs may make compression a more viable option for optimizing memory bound applications, and network bound applications on computing clusters.

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.

Attia, O. G., Johnson, T., Townsend, K. R., Jones, P. H., and Zambreno, J. (2014). CyGraph: a reconfigurable architecture for parallel breadth-first search. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium and PhD Forum (IPDPSPW)*, pages 228–235.

Bachir, T. O. and David, J.-P. (2010). Performing floating-point accumulation on a modern FPGA in single and double precision. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 105–108.

Banescu, S., Dinechin, F. D., Pasca, B., and Tudoran, R. (2010). *ACM SIGARCH Computer Architecture News*, 38(4):73–79.

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.

Buluç, A., Williams, S., Oliker, L., and Demmel, J. (2011). Reduce-bandwidth multithreded algorithms for sparse matrix-vector multiplication. pages 721–733.

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.

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

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.

Davis, T. A. and Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1).

de Dinechin, F., Nguyen, H. D., and Pasca, B. (2010). Pipelined FPGA adders. pages 422–427.

de Dinechin, F. and Pasca, B. (2011). Designing custom arithmetic data paths with FloPoCo. *IEEE Design and Test of Computers*, 28(4):18–27.

de Dinechin, F., Pasca, B., Cret, O., and Tudoran, R. (2008). An fpga-specific approach to floating-point accumulation and sum-of-products. In *Proceedings of the International Conference on Field-Programable Technology (FPT)*, pages 33–40.

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

Deutsch, P. L. (1996). GZIP file format specification version 4.3.

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203.

Engelson, V., Fritzson, D., and Fritzson, P. (2000). Lossless compression of high-volume numerical data from simulations. In *Proceedings of the Data Compression Conference (DCC)*, pages 574–586.

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.

Fowers, J., Ovtcharov, K., Strauss, K., Chung, E. S., and Stitt, G. (2014). A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 36–43.

Gerards, M. (2008). Streaming reduction circuit for sparse matrix vector multiplication in FPGAs. Master's thesis, University of Twente.

Goeman, B., Vandierendonck, H., and Bosschere, K. (2001). Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the IEEE*

International Symposium on High Performance Computing Architecture (HPCA), pages 207–216.

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

Good, T. and Benaissa, M. (2006). Very small FPGA application-specfic instruction processor for AES. *IEEE Transactions on Circuits and Systems—Part I: Fundamental Theory and Applications*, 53(7):1477–1486.

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

Grigoras, P., Burovskiy, P., Hung, E., and Luk, W. (2015). Accelerating SpMV on FPGAs by lossless nonzero compression. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 64–67.

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. volume 40, pages 1098–1101.

Im, E. (2000). *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, University of California, Berkeley.

Johnson, K. T., Hurson, A. R., and Shirazi, B. (1993). General-purpose systolic arrays. In *IEEE Computer*, pages 20–31.

Jones, A. K., Hoare, R., Kusic, D., Fazekas, J., and Foster, J. (2005). An FPGA-based VLIW processor with custom hardware execution. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programable Gate Arrays (FPGA)*, pages 107–117.

Kalokerinos, G., Papaefstathiou, V., and Nikiforos, G. (2009). FPGA implementation of a configurable cache/scratchpad memory with virtualized user-level RDMA capability. In *Proceed-*

ings of the IEEE International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), pages 149–156.

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

Ke, J., Burtscher, M., and Speight, E. (2004). Runtime compression of MPI messages to improve the performance and scalability of parallel applications. In *Proceedings of the IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 59–65.

Kestur, S., Davis, J. D., and Chung, E. S. (2012). Towards a universal FPGA matrix-vector multiplication architecture. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16.

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.

Lindstrom, P. and Isenburg, M. (2006). Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 12(5):1245–1250.

Lipasti, M. H., Wilkerson, C. B., and Shen, J. P. (1996). Value locality and load value prediction. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 138–147.

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.

Monakov, A. (2012). Specialized sparse matrix formats and SpMV kernel tuning for GPUs. In *Proceedings of the GPU Technology Conference (GTC)*.

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. (2009). A high-performance double precision accumulator. In *Proceedings of the International Conference on Field-Programable Technology (FPT)*, pages 500–503.

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.

Nelson, C., Townsend, K. R., Rao, B. S., Jones, P. H., and Zambreno, J. (2012). Shepard: A fast exact match short read aligner. In *Proceedings of the IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 91–94.

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.

Nishtala, R., Vuduc, R. W., Demmel, J. W., and Yelick, K. A. (2007). When cache blocking of sparse matrix vector multiply work and why. 18(3):297–311.

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

Ratanaworabhan, P., Ke, J., and Burtscher, M. (2006). Fast lossless compression of scientific floating-point data. In *Proceedings of the Data Compression Conference (DCC)*, pages 133–142.

Richter, T. (2009). Evaluation of floating point image compression. In *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, pages 1909–1912.

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.

Sato, D., Xie, Y., Weiss, J. N., Qu, Z., Garfinkel, A., and Sanderson, A. R. (2009). Acceleration of cardiac tissue simulation with graphic processing units. *Medical and Biological Engineering and Computing (MBEC)*, 47(9):1011–1015.

Sazeides, Y. and Smith, J. E. (1997). The predictability of data values. In *Proceedings of the ACM/IEEE International Symposium on Microarchitectures (MICRO)*, pages 248–258.

Schindler, M. (1998). A fast renormalisation for arithmetic coding. In *Proceedings of the Data Compression Conference (DCC)*, page 572.

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

Shan, Y., Wu, T., Wang, Y., Wang, B., Wang, Z., Xu, N., and Yang, H. (2010). FPGA and GPU implementation of large scale SpMV.

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

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

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

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.

Townsend, K. R. and Zambreno, J. (2015). A multi-phase approach to floating-point compression.

Umuroglu, Y. and Jahre, M. (2014). An energy efficient column-major backend for FPGA SpMV accelerators. In *Proceedings of the IEEE Internation Conference on Computer Design (ICCD)*, pages 432–439.

Umuroglu, Y. and Jahre, M. (2015). A vector caching scheme for streaming fpga spmv accelerators. In *Proceedings of the ACM International Workshop on Applied Reconfigurable Computing (ARC)*, pages 15–26.

Vangal, S., Hoskote, Y., Borkar, N., and Alvandpour, A. (2006). A 6.2-GFlops floating-point multiply-accumulator with conditional normalization. *IEEE Journal of Solid-State Circuits (JSSC)*, 41(10):2314–2323.

Vuduc, R. W. and Moon, H.-J. (2005). Fast sparse matrix-vector multiplication by exploiting variable block structure. In *Proceedings of the IEEE International Conference on High Performance and Communications (HPCC)*, pages 807–816.

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.

Wang, Z., Qiu, X., Zhang, A., Cheng, Y., Peng, Y., and Sun, S. (2015). Yet another hybrid strategy for auto-tuning SpMV on GPUs. 9(5):97–106.

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

Woods, D. (2009). Coherent shared memories for FPGAs. Master's thesis, University of Toronto.

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.

Zeng, T., Townsend, K. R., Duan, J., and Chen, D. (2013). A 15-bit binary-weighted current-steering DAC with ordered element matching. pages 1–4.

Zhang, Y., Shalabi, Y. H., Jain, R., Nagar, K. K., and Bakos, J. D. (2009). FPGA vs. GPU for sparse matrix vector multiply. pages 255–262.

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