ACCELERATION OF SPARSE MATRIX MULTIPLICATION USING BIT-SERIAL ARITHMETIC

A Thesis
Presented to
The Academic Faculty

By

Matthew Denton

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

August 2021

ACCELERATION OF SPARSE MATRIX MULTIPLICATION USING BIT-SERIAL ARITHMETIC

Thesis committee:

Dr. Tushar Krishna
Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Arijit Raychowdhury
Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Moinuddin Qureshi
Computer Science
Georgia Institute of Technology

Date approved: July 30, 2021

The master has failed more times than the beginner has tried.

Stephen McCranie

Dedicated to Kentaro Miura, author of *Berserk*

ACKNOWLEDGMENTS

I would like to thank Tushar Krishna and Herman Schmit for guiding my growth as a researcher in computer architecture, giving me extensive exposure to both industry and academic research.

I'm honored and humbled to have worked with Sudhakar Yalamanchili and Moinuddin Qureshi, who showed me that great researchers can also be great teachers and compassionate people.

A huge thank you to Peter Matheu and Amir Yazdanbakhsh at Google for feedback on the direction of this work.

I would be remiss not to acknowledge the Synergy Lab for thought provoking conversations and great times in the lab. In no particular order, I would like to thank Mayank Parasar, Hyoukjun Kwon, Ananda Samajdar, Saeed Rashidi, Eric Qin, Sheng-Chun (Felix) Kao, Yehowshua Immanuel, Vineet Nadella, Ankit Arora, Geonhwa Jeong, and William (Jonghoon) Won for all the great times.

A large portion of this work was done as an internship project with the Google Brain team. The author gratefully acknowledges the support for this work offered by Google Research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Machine Learning inference requires the multiplication of large, sparse matrices. We argue that direct spatial implementation of these fixed matrices minimizes the work performed in the computation, and allows for significant reduction in latency and power through constant propagation and logic minimization. Bit-serial arithmetic enables massive static matrices to be implemented. We present the structure of our bit-serial matrix multiplier, and evaluate using canonical signed digit representation to further reduce logic utilization. We have implemented these matrices on a large FPGA and provide a cost model that is simple and extensible. These FPGA implementations, on average, reduce latency by 50x up to 86x versus GPU libraries. Comparing against a recent sparse DNN accelerator, we measure a 4.1x to 47x reduction in latency depending on matrix dimension and sparsity. Throughput of the FPGA solution is also competitive for a wide range of matrix dimensions and batch sizes. Finally, we discuss ways these techniques could be deployed in ASICs, making them applicable for dynamic sparse matrix computations.

CHAPTER 1
INTRODUCTION

Machine Learning (ML), particularly Deep Neural Networks (DNNs) are seeing increased use in solving complicated tasks in computer vision[1], natural language processing [2], game playing [3], and other domains.

When querying these networks, the primary computational primitive invoked is the matrix-multiply[4], which has seen sparsity[5] as an increasingly common property of such matrices.

The computer architecture community has had great strides in accelerating matrix multiply[6], but fundamentally designs are still limited by sparse indexing and tiling, which are techniques that essentially map a large sparse matrix multiply into many dense matrix multiplications, which make them more efficient.

1.1 Goals

The goal of this work is to minimize the cost of performing sparse matrix multiply by building the required arithmetic from the ground up in a way which nearly eliminates sparse indexing and tiling. We can do this by building the matrix-multiply as a circuit in programmable hardware, which allows us to perform logic minimization and constant propagation in the hardware implementation itself instead of introducing transforms which map sparse computations into dense ones. We begin with a review of the rise of deep learning, and how computational power has driven much of it.

1.2 The Rise of Deep Learning in the Modern Era

Deep Learning (DL) is a form of machine learning which involves approximating functions using deep neural networks. Many researchers point to the ImageNet 2012 challenge as the dawn of modern deep learning, where AlexNet [7] won the competition by a wide margin using a convolutional neural network. After AlexNet, deep learning began to become more and more pervasive in solving computational problems. These methods were also applied to natural language processing, accomplishing tasks such as language translation. [8] The world champion of the game GO was defeated by a machine for the first time by a system called AlphaGO. [3]

1.2.1 Accelerating Machine Learning through Custom Hardware

Much of the AlexNet moment and resurgence of deep learning is attributed to availability of parallel compute, at that time on GPUs. While GPUs are fantastic general purpose parallel machines, the end of Moore's law has ushered in efforts from computer architects to design Application Specific Integrated Circuits also know as ASICs or "accelerators." For DNN's, the primary compute primitive is matrix-multiply. [4]

In academia, one of the early DNN accelerators was DianNao[9], which focused on convolutions. The field began to gain momentum when Google announced that they had built such an accelerator for datacenter inference, which they called Tensor Processing Unit (TPU).[6] At the heart of TPU was, as expected, a very fast matrix multiplier. Following this, NVIDIA introduced small matrix multipliers in their Volta architecture, which they dubbed "Tensor Cores". [10]

1.2.2 Tiling a Matrix Multiply

When composing a matrix multiplier, there will always be cases where the desired multiply does not fit in hardware, and must be accomplished by breaking the multiply down into multiple smaller matrix multiplies that do fit in hardware. We call this ”tiling,” and it’s one of the most expensive operations in DNN acceleration. In fact, how a multiply should be tiled is an active area of research. [11]

As an example, consider a systolic array matrix multiplier as shown in Figure 1.1. It consists of a 2D array of MACs (Multiply-And-Accumulate), which are connected to their neighbors. This is not a fully featured or fleshed out design, but rather a depiction to aid the reader in understanding. We consider an output stationary dataflow, where the output accumulator for each element in the resulting matrix is still at a given position in the array. We stream the input rows across this array, and the weight columns down the array. Each MAC multiplies it’s two inputs, and then forwards the row input to the right and the column input down. After all the rows and columns are done streaming, the resulting matrix lives in the array itself. Depicted here is a 4x4 systolic array multiplying two 4x4 matrices together.

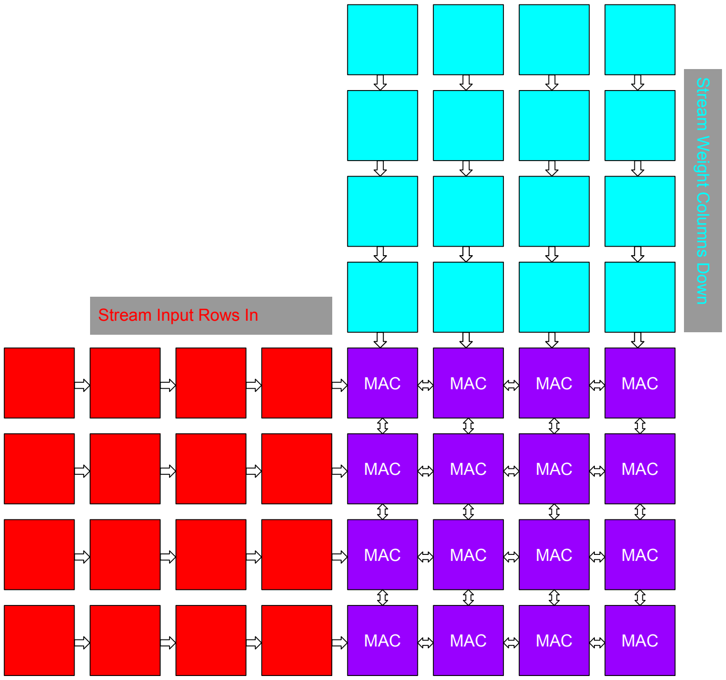


Figure 1.1: Systolic

Now consider a more constrained scenario, where we only have a 2x2 systolic array, but we want to accomplish the same 4x4 matrix multiply, as shown in Figure 1.2

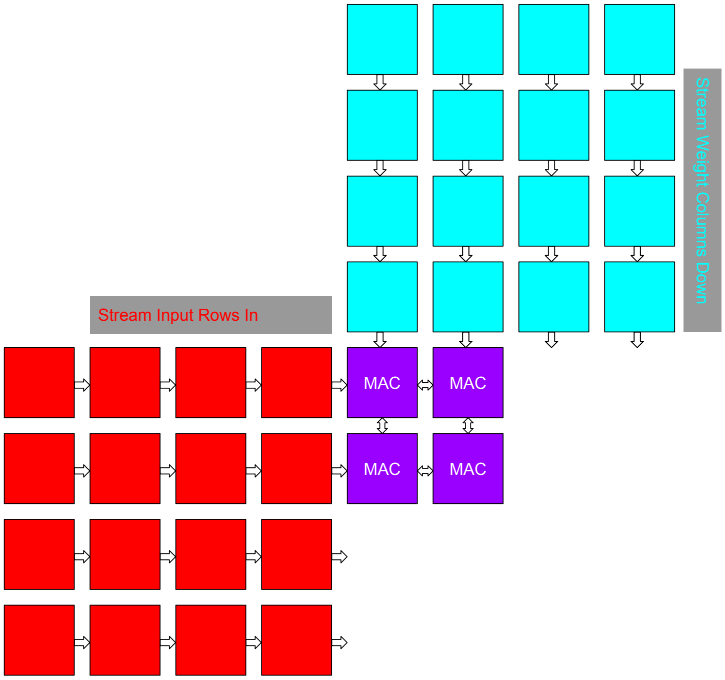


Figure 1.2: Systolic Matrix Multiply with constrained hardware

In this scenario, we must tile the multiply as mentioned earlier. Let's walk through that example. Figure 1.3 shows a 4x4 matrix multiply being broken down into 4 2x2 multiplies, which are done across time reusing the systolic functional unit. In this case, it not only takes more time, but we need to load each input and weight element twice to stream it through the array twice, which causes thrashing to the memory system and complicates memory accesses. Furthemore, the smaller matrix multiply leaves some memory unused as shown in black.

We again stress that this is a toy example, but the high-level idea is that all matrix-multiply units (whether that be TPU, GPU, CPU, etc) have some capacity. When that capacity is exceeded, tiling the matrix-multiply is costly from a time and memory perspective.

1.2.3 Sparse Indexing

In an effort to reduce the compute and memory needed to perform evaluation of the DNNs, sparsity was introduced to them. [5] Sparsity refers to a portion of matrix entries being 0. When an entry is 0 it does not contribute to the inner product, and therefore that multiply can be skipped.

It's simple to transform a sparse dot-product into a smaller dense one, using sparse indexing. We can store the indices of all the dense elements in a matrix, and then use those indices to compute the multiply. However, in current accelerator paradigms, this indexing is not free. We have to encode the matrix with some meta-data, which then must be processed by the accelerator to form the multiply. This costs both memory and compute, so there is some optimal tradeoff point. An example is shown in Figure 1.4. We take the weight matrix and encode the location of non-zero elements. When the input comes in, we only load the elements associated with those non-zero elements. This reduces the memory footprint and compute, but comes with overhead. Again, this is a simple example of the myriad of ways there are to do sparse dot-product (and by extension sparse matrix-multiply).

1.2.4 Quantization and Bit-serial Arithmetic

Finally, and particularly relevant to this work, is that DNNs have been shown to work as well when the entries are quantized. The DNN is initially trained using floating-point representation, but then for inference we can use integers. There has been some work on using bit-serial arithmetic to perform integer matrix-multiply more efficiently, which we build on here. [12]

1.3 Related Work

DNN Accelerators

In this work we focus on the matrix-multiply in DNNs, which has been the focus of many ground-breaking architectures in this field. Google's TPU [6] presented a systolic-array for matrix multiply which involved a 2D grid of processing elements with neighbor-to-neighbor communication. Eyeriss[13] focused on convolutions as a primitive, which allowed them to exploit data reuse in the convolution filter process. They introduced a taxonomy of dataflows for accelerators, and presented row-stationary as a key dataflow for convolutions and flexible architectures. EIE [14] encoded sparse DNNs into an intermediate representation, and then decodes this in real time to perform efficient sparse inference. SIGMA[15] adapts the flexible dataflow and reduction trees of MAERI [16], adding in sparsity support through forwarding adder networks.

Bit-serial accelerators

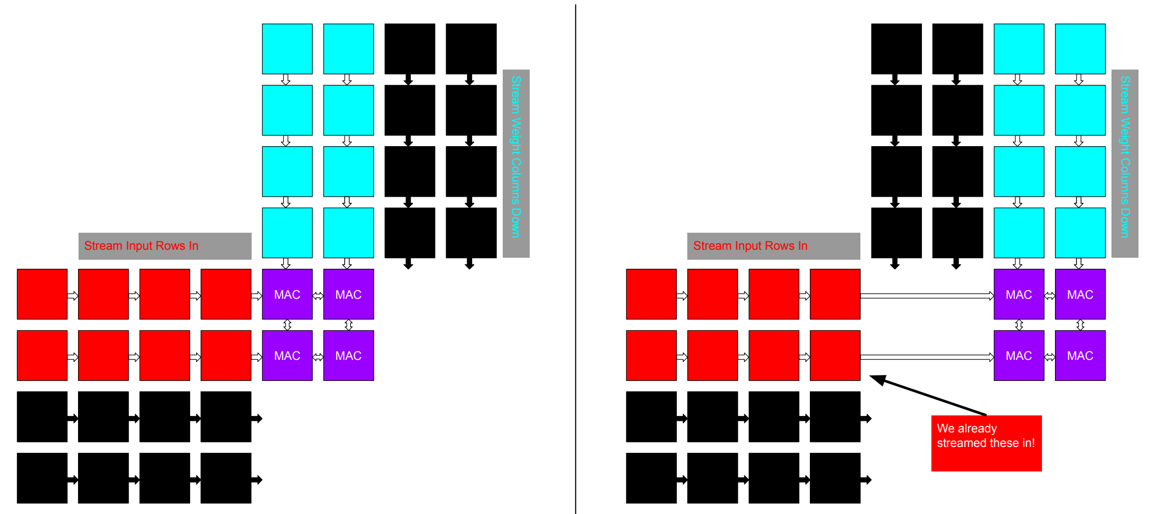
Bit-serial arithmetic was an important technique decades ago when parallel arithmetic was too costly. Simple VLSI implementations of bit-serial neural networks [17] were proposed as far back as 1988. More recently, DNN accelerators have used bit-serial arithmetic to support variable precision arithmetic. Essentially, each layer of a DNN has a differing range of weight values, and it's unecessary to encode all weights with the same bitwidth. STRIPES [18] presents a bit-serial architecture that enables different precisions on a per-layer basis, quantizing each layer to their required bit-widths. It saves energy by storing and processing only the widths needed, whether narrow or wide. Bit-fusion [19] enables dynamic-precision for inputs as well. Bit-pragmatic [20] and Laconic [12] exploit bit-level sparsity. They detect the bits that are zero and skip those multiplications. In all the above cases, the DNN dimensions often exceed the compute capability of the hardware, so tiling and striding the inner product is regularly employed.

1.4 Summary

We have introduced the matrix-multiply as the key compute kernel in DNN's, where sparsity is increasingly common. Performing these matrix-multiplies on hardware involves sparse indexing and tiling, which are both inefficient. We reviewed the literature of DNN accelerators as well as specifically bit-serial accelerators. Next, we introduce our novel architecture which transforms a weight matrix into a representative circuit.

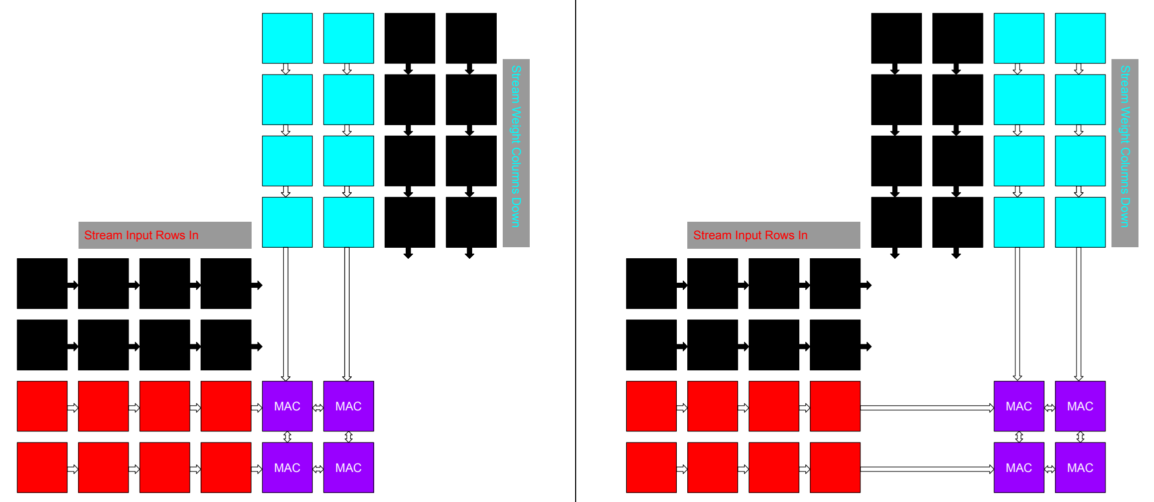
The content of this thesis is organized as follows

- Chapter 1 introduced Deep Learning workloads as matrix-multiply and demonstrates the limitations of tiling and sparse indexing
- Chapter 2 presents a novel architecture for implementing a sparse integer matrix-multiply in hardware directly, and evaluates the mapping of this design onto an FPGA
- Chapter 3 evaluates the performance of this FPGA implementation against a GPU and DNN accelerator
- Chapter 4 discusses how a CGRA variant of this architecture could be built which would yield even higher performance
- Chapter 5 concludes



We already streamed these in!

a) First tile b) Second tile



c) Third tile d) Fourth tile

Figure 1.3: A matrix multiply is broken down into tiles, which must be done independently across time. Notice that each input row and weight column is streamed across the systolic array twice as opposed to once in the non-tiled approach. Black squares represent elements in memory we are not using for that tile

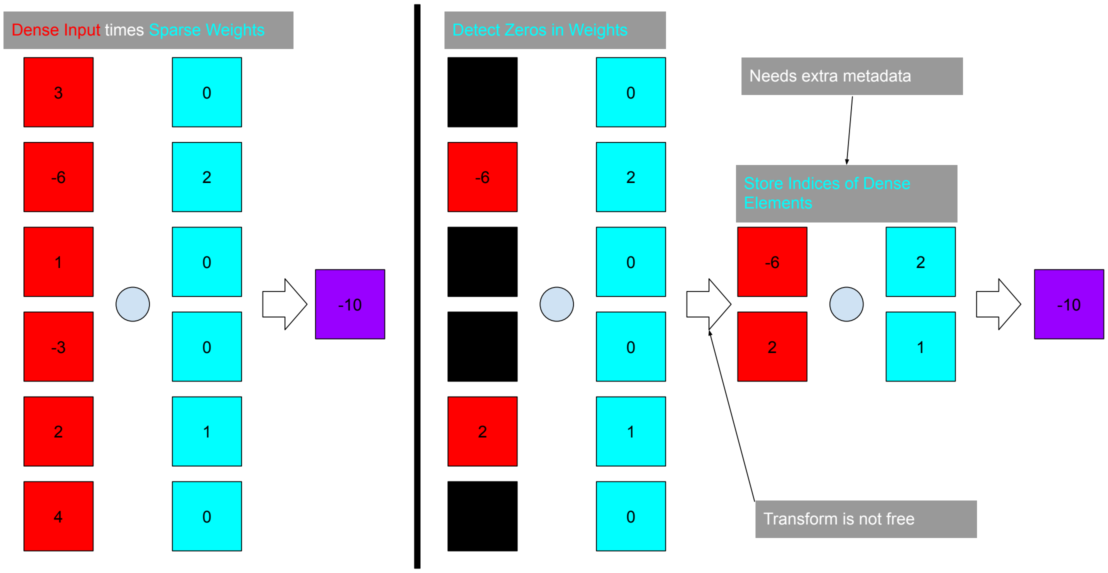


Figure 1.4: Sparse dot-product. The left shows the un-encoded dense multiply. The right transforms this into a simpler operation using encoding.

CHAPTER 2

BASELINE FPGA ARCHITECTURE

In this chapter, we present out novel architecture for sparse matrix-multiply and implement it on an FPGA. As discussed in Chapter 1, machine learning uses matrix multiples to build the majority of the compute graph. These weight matrices are both sparse and large, and can be quantized. Our goal is to take a weight matrix and implement it as a circuit in programmable hardware. Specifically, we want to implement the relation seen in Equation 2.1

$$o = a^{\intercal} V \tag{2.1}$$

where V is a sparse integer weight matrix, and a is in input vector. This primitive is commonly called SpMV, and can be used to build SpMM, both of which are the basis of many deep learning kernels.

The properties of V allow us to optimize the logic implementation by performing constant propagation on the matrix to remove the logic corresponding to all zeros; both zeros as whole terms, and bits that are zeros within individual terms. It would be possible to implement these matrices in custom logic (ASIC), but in this work we use an FPGA. The FPGA consists of an array of Lookup Tables (LUTs), Registers (FFs), and programmable interconnect. FPGAs generally include other programmable resources such as embedded multipliers and SRAMs, which we will not consider in this work. The particular FPGA we are using has the capability to re-purpose some of the LUTs into small RAMs or shift registers which are called LUTRAMs.

Our FPGA design flow takes the content of the matrices and compiles it to a physical design consisting of the programmable logic resources and interconnect. This design flow produces an achievable frequency, area, and power estimation.

2.1 The Microarchitecture

Figure 2.1 shows the design of a bit-serial adder, which is the basic building block of bit-serial arithmetic units.

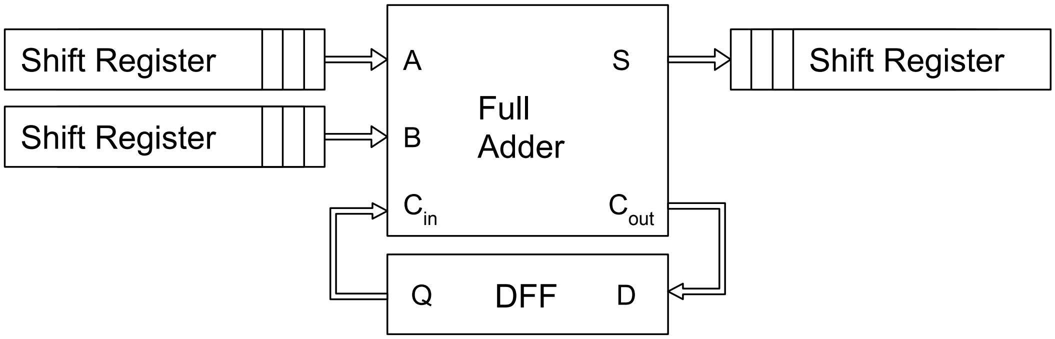


Figure 2.1: Bit-serial adder

In the bit-serial adder, inputs A and B are shifted into a full-adder one bit each clock cycle, LSb first. The result of the addition (S) is shifted into a result register. The carry is registered as the carry input for the next cycle. This is analogous to manual long addition - add one digit at a time, with any resulting carry being added in the next most significant digit. Rather than chaining the carries through multiple adders ”in space”, we use a single adder and process the carry ”in time”. Table Table 2.1 shows an example of a bit-serial addition of $3_{10} + 7_{10} = 10_{10} \leftrightarrow 011_2 + 111_2 = 1010_2$

Table 2.1: Bit-serial Addition Example

Cycle	C_{in}	A	B	S	C_{out}	Result
1	0	1	1	0	1	0000
2	1	1	1	1	1	1000
3	1	0	1	0	1	0100
4	1	0	0	1	0	1010

Similarly, we can create a bit-serial subtractor that performs $a - b$ by initializing the carry bit to 1, and adding a NOT gate between b’s register and the full adder. This scheme is equivalent to negating b in two's complement and then adding it to a. In the FPGA, the bit serial adder or subtractor can be mapped to a single 6-input LUT and two registers. The timing path that determines the frequency for this design (the critical path) will be very short: a single stage of logic, the setup to the flop, and any interconnect delay between the components.

Single-bit dot-product Using these primitives, we now build a single-bit dot-product, as described below:

$$s = \sum_{i=1}^{N} a_i \times b_i \tag{2.2}$$

Where a, b are N-length vectors whose elements are 1-bit wide, and s is a scalar integer. Because we are multiplying single bits, we can realize the multiplication with a simple AND gate. After multiplying each pair of elements, we reduce the partial sums through a tree of bit-serial adders.

Multi-bit input We can extend this design to support one of the operands, a, being multiple bits wide. We hold the single-bit vector b fixed, and stream the multi-bit vector through the circuit one bit at a time from LSb to MSb using a shift register for each dimension. Similarly, we capture the result in a shift register. One can think of this circuit as summing the elements of a that are selected by a Boolean vector b. This design is shown in Figure 2.2a.

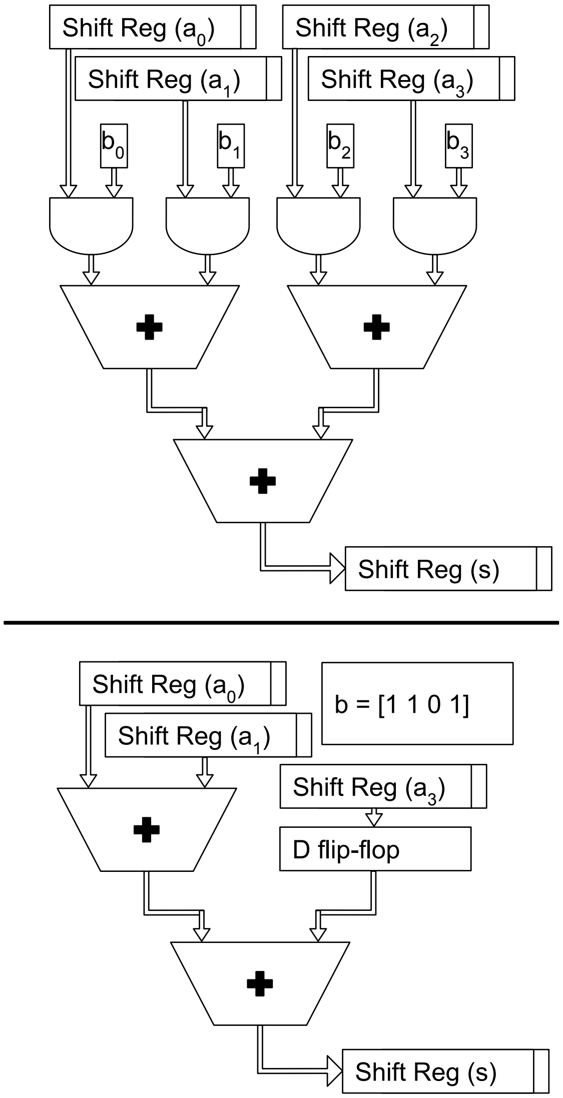


Figure 2.2: a)Multi-bit by fixed single-bit dot-product with tree reduction and shift registers (N=4) (trapezoids represent bit-serial adders, which have flip-flips between each stage but are omitted for brevity)
b) Setting b = [1 1 0 1] reduces the circuit

Note that because this is merely summing the dimensions selected by b, this circuit works for both signed and unsigned a inputs. To ensure signed inputs produce the correct sign bit, we sign extend the input a from the shift register until the computation has finished.

Given that our boolean vector b is fixed, consider a given element b_i. If this element is set to 1, the input to the AND gate is 1, so the output of the AND gate will just be the associated a_i. In this case, we can cull the AND gate entirely and connect the input a_i directly to the bit serial adder. On the other hand, if the element is 0, the AND gate will have a fixed 0 input and therefore its output will always be 0. Furthermore, one of the inputs to the bit-serial adder is now a fixed 0, so the adder is just passing the other value forward. In this case, the adder is acting as a D-flip-flop. This means we can not only cull the AND gate, but we can also replace the adder with a single flip-flop, which reduces the cost significantly. Put simply, we can greatly reduce the cost of this circuit by fixing b, and the cost should be proportional to the number of bits set in b. **This optimization is the fundamental minimization technique in our design.** An example of this reduction can be seen in Figure 2.2b. We will revisit this idea in Section section 2.2.

Multi-bit input and weight Finally, we can combine multiple of these multi-bit by 1-bit dot-products to create a full multi-bit dot-product. This allows the dot-product of two integer vectors, each with its own bit-width. Because each dot-product in the previous circuit occurs at the bit-level in b, we merely need to create such a circuit for each bit position in b, and then combine the results for each bit position.

Luckily, the bit-serial adders already have this capability by shifting in time. For example, to make a 2-bit dot-product circuit, we make one circuit for the MSb, and one for the LSb. The results of the two are combined by feeding the LSb result into a bit-serial adder, and then feeding the MSb result, delayed by 1 cycle, into the same bit-serial adder, which then outputs into a shift register to capture the final result. Delaying the MSb result by one cycle effectively multiplies it by 2. To extend this idea to arbitrary bit-width, we just create a chain of bit-serial adders, where the bottom input is the previous link in the chain, and the top input is the corresponding bit. In this case, the result of a bit position is delayed accordingly. A multi-bit example is shown in Figure 2.3. Notice that the MSb is fed into a bit-serial adder along with 0, which becomes a D flip-flop.

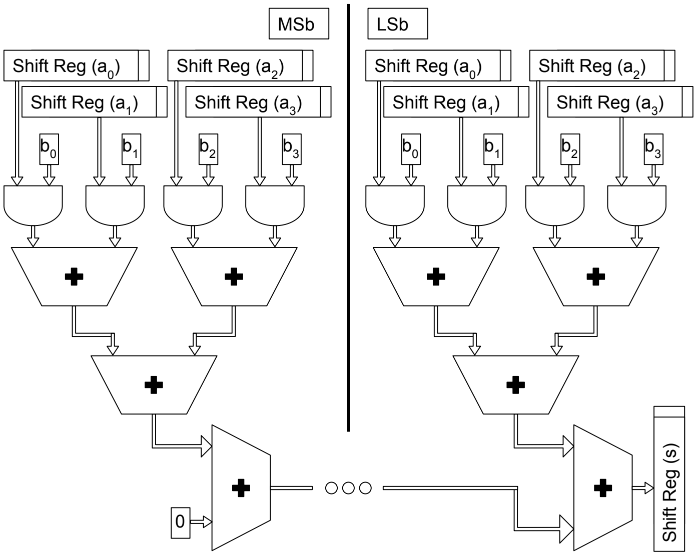


Figure 2.3: Multi-bit Dot-product

This architecture works for signed inputs and unsigned inputs, but the weights are unsigned. An easy way to implement signed weights is to separate the positive and negative terms of the b vector into two separate unsigned vectors, and simply subtract the two resultant streams. Because the number of ones in the two matrices is conserved by this transform, it makes almost no impact on the total area, and adds a single cycle to the latency.

Vector-matrix product Now that we have a unit to perform dot-products, we can complete this matrix-multiplier by creating a dot-product unit for each column in the matrix. Then, we just broadcast the input on each row to all columns, and that results in a matrix multiplier, which takes shape as shown in Figure 2.4. (Full bit-serial circuits are omitted here for brevity)

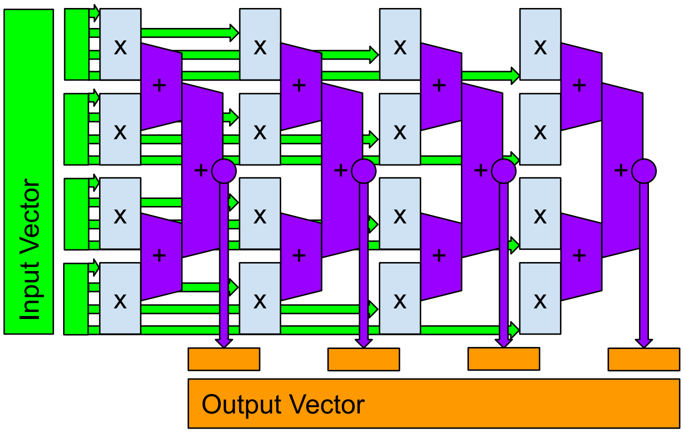


Figure 2.4: Full Vector-Matrix Multiplier Architecture
Broadcast of inputs shown in green. Multiplication at bit-level of fixed weights shown in blue. Per-column partial sum reduction shown in purple, yielding the final output vector in orange

Because the computations are parallelized across the matrix columns, and then again across each bit-position, the latency for this design is described in Equation 2.3:

$$\text{Latency} = BW_i + BW_w + \log_2 R + 2 \tag{2.3}$$

Where BW_i and BW_w are the bitwidth of the input and weights respectively, and R is the number of rows in the matrix. We incur the input width to stream the input in, the output width to stream the output out, and our adder tree is logarithmic in depth. We incur a single cycle to accumulate across bit positions and an additional cycle to subtract the positive and negative weight matrices. For example, given 8-bit inputs and weights and a 1024x1024 weight matrix, we perform the vector-matrix product in $8+8+\log_2(1024)+2 = 28$ cycles.

2.2 RTL Synthesis Results

In this section, we seek to understand the behaviour of our design by studying it on small matrices. We extend the insights here to larger matrices in Section section 2.4. Recall the optimization from Section section 2.1 that if a bit in the weight-matrix is 0, we can cull the AND gate and bit-serial adder, replacing it with a simple D flip-flop. We coded our design in SystemVerilog and ran synthesis in Xilinx Vivado 2020.2 targeting Xilinx UltraScale+ devices [21]. If our design behaves as expected, the hardware cost should be proportional to the number of bits set in the weight matrix. To test this, we randomly initialized a series of 64x64 weight matrices at 8-bit precision. For each bit in the weight matrix, we sample from a Bernoulli distribution, where the p parameter is equal to (1 - bit_sparsity). That is to say, the bit-sparsity of the weight matrix is the number of bits that are 0 out of the total number of bits. 0% bit-sparse means all bits are 1, 50% means the bits are uniformly random between 0 and 1, and 100% means all bits are 0. This gives us a straightforward way to measure the cost of these matrices.

Figure 2.5 shows synthesis results for sweeping bit-sparsity from 0% to 100%. We report utilization of LUTs, FFs, and LUTRAMs.

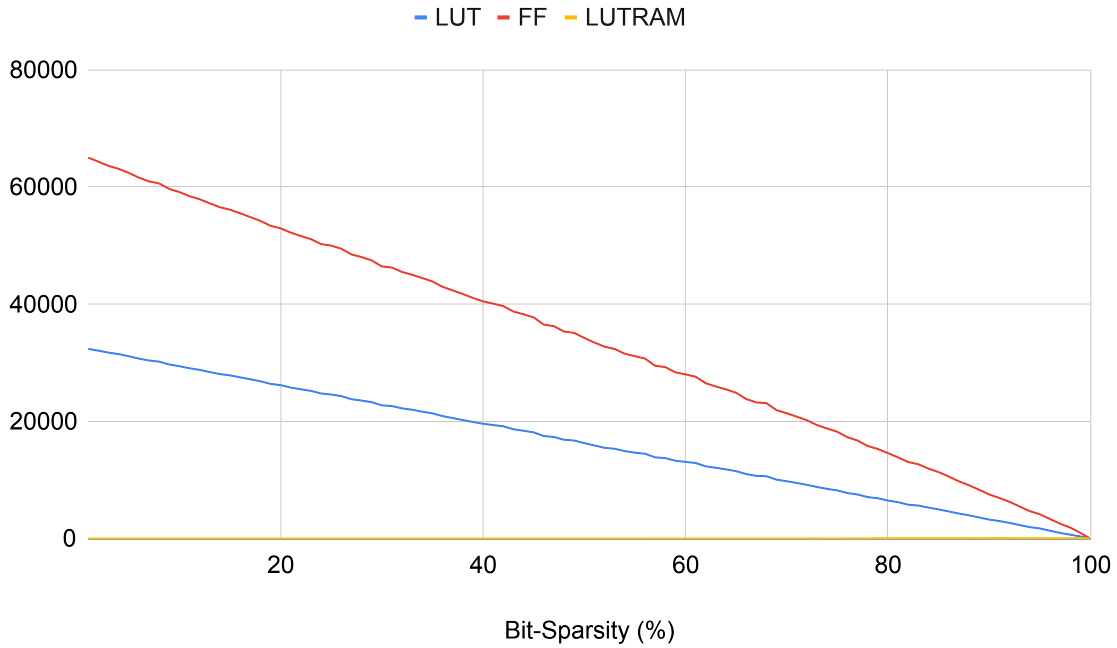


Figure 2.5: Hardware utilization vs bit-sparsity of a 64x64 matrix

As expected, the total hardware cost of our architecture is linear with respect to the number of bits set in the weight matrix.

This illustrates that our architecture effectively exploits bit-level sparsity to reduce the cost of fixed-matrix multiplication. Recent advances in neural-networks and architecture have similarly sought to exploit element-level sparsity to make computations more efficient. [22, 23] By element-level sparsity, we mean a portion of the weights being 0, so there is no need to multiply by them. Because bit-level sparsity is a super-set of element-level sparsity, we wanted to know if our design was ineffective given an element-sparse matrix. To test this, we randomly initialized a set of matrices, where the weights are sampled from a uniform distribution of all possible values for the given bit-width. In this case, the matrix is 50% bit-sparse, as every bit has an equal probability of being 0 or 1. We then randomly replace matrix elements with 0 until we reach a desired level of element-sparsity. Taking these samples, we convert the element-sparse value into a bit-sparse value, and compare the two approaches. This second experiment encourages bits to gather in individual elements, rather than the bit-sparse experiment which encouraged bits to be spread out. Figure 2.6 shows our findings and plots them against the original experiment.

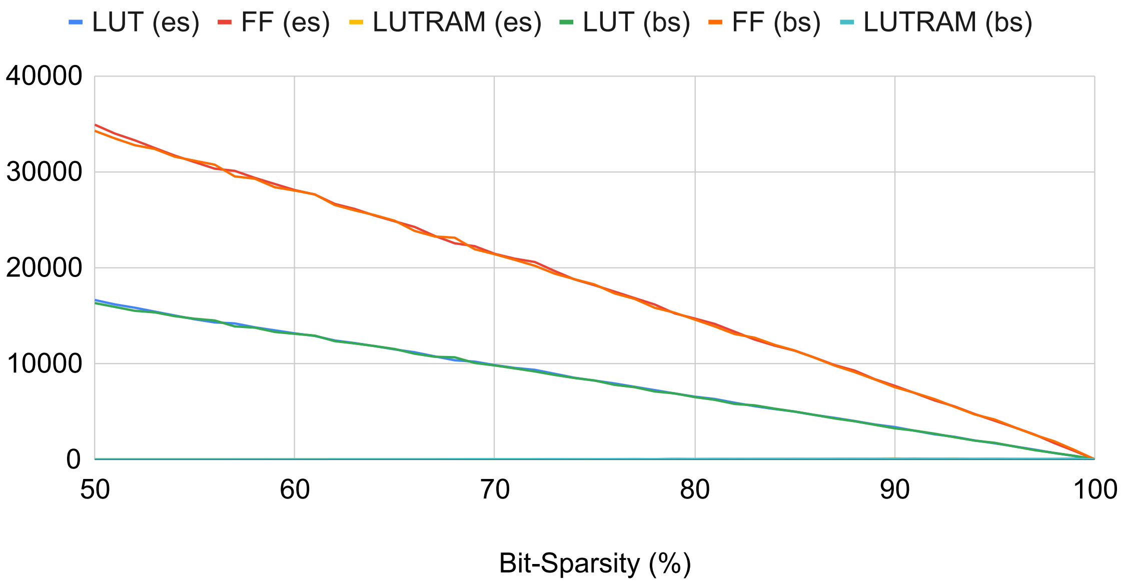


Figure 2.6: Cost of Element Sparse Matrix Compared To Bit-Sparse Matrix
Here, (es) and (bs) designate Element Sparse and Bit-Sparse schemes respectively. While results do not match exactly, they are within an acceptable noise margin

The graph shows us that it doesn't matter if the bits are concentrated or not. The two lines are nearly identical, which means that we don't have to make any concessions to support element-sparse designs. Unlike most sparse accelerators, this design exploits sparsity in the elements when matrix entries are zero, and also elicits great benefit from elements being powers-of-two.

Figure 2.7 shows the LUT and FF utilization vs matrix size. The cost is quadratic with respect to matrix dimension and therefore linear with respect to the number of elements. This implies there is little optimization our RTL flow is doing across weight elements. In other words, large matrices are no more and no less dense than smaller matrices.

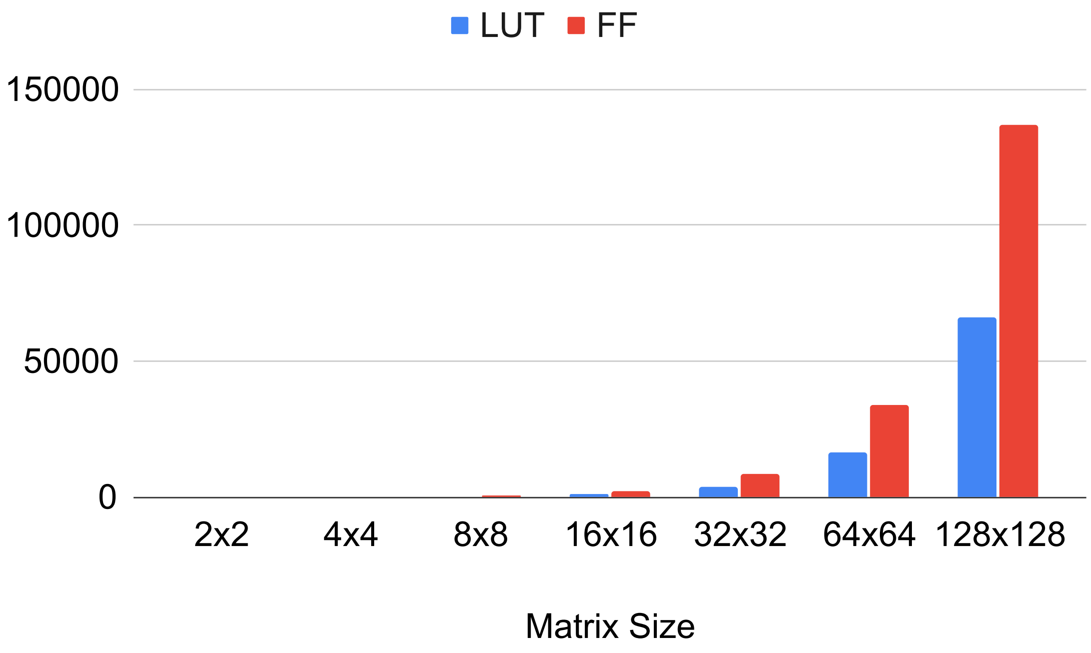


Figure 2.7: Hardware Utilization vs Matrix Size for random 8-bit integers

We also explore the cost function with respect to the bit-width of the weights. Figure Figure 2.8 shows the cost of a 64x64 matrix, sweeping the weights bit-width from 1-bit to 32-bit. Because the architecture performs 1-bit dot-products for each bit-position, and then accumulates the results, we would expect there to be little cross-bit optimizations. Indeed, we observe a linear LUT and FF cost with respect to the bit-width of the weights.

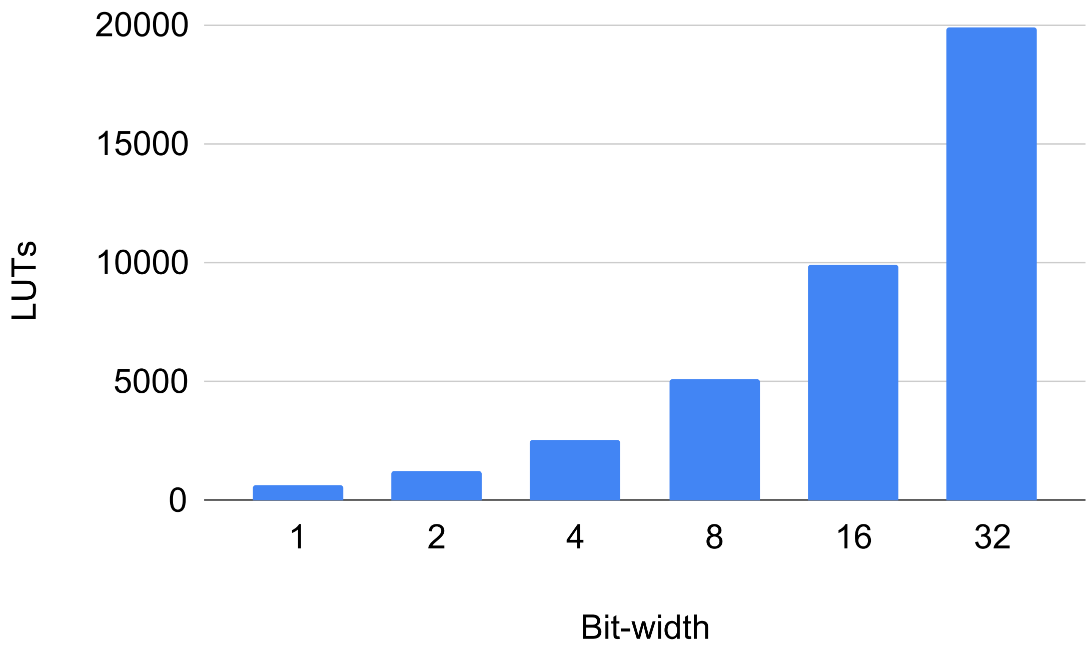


Figure 2.8: Hardware Utilization of 64x64 Random Matrix for Varying Bitwidths

2.3 Canonical Signed Digit

In an effort to reduce the cost of the multiplier, we consider Canonical Signed Digit (CSD) representations [24]. CSD decomposes an unsigned integer into a sum of a positive and negative integers, where the positive and negative integer together have fewer set bits than the original number. For example, consider the following :

$$15_{10} = 16_{10} - 1_{10} \leftrightarrow 1111_2 = 10000_2 - 00001_2$$

Notice that we can decompose a number that had four bits set to the difference of two numbers which each have one bit set, for a total of two. Do also take note that the bit-width of the decomposition is one wider than the original. Recall that the cost function of our multiplier is the number of bits set, so this shows promise in our case. We transform

Equation 2.1 using:

$$V \equiv P - N \implies o = a^{\intercal}(P - N) = (a^{\intercal}P) - (a^{\intercal}N) \tag{2.4}$$

To transform V into N and P, we apply the algorithm in Listing 2.1 element-wise, placing positive elements in P, and placing the absolute value of negative elements in N. In short, the algorithm searches for strings of consecutive 1 bits, which we call a chain. If a chain is length 1, nothing is done. If a chain is length 2, we flip a coin. On heads, we replace the chain with a +1 in the bit position one-past that of the MSb in the chain and a -1 at the LSb in the chain. On tails, we leave the original representation. For chains of length 3 and greater, we perform the same replacement that we do for heads on length 2. We introduce the random variable to balance the decomposition, since a transformation of a length 2 chain has no benefit and no detriment.

When operating on signed weights, we perform a CSD transform on both the positive and negative weight matrices. Positive elements that result from CSD remain in the original matrix, and negative elements are transferred to the opposite weight matrix.

```
def convert_to_csd(num_bin_list):
  local_list = list(num_bin_list)
  target = [0] * (len(local_list) + 1)
  local_list.reverse()
  chain_start = -1  # are we in a chain?
  for i in range(len(target)):
    if i < len(local_list):
      bit = local_list[i]
    else:
      bit = 0
    if bit == 0:
      if chain_start == -1:  # no chain
        target[i] = 0  # nothing to be done here
      else:
        #  We terminate a chain, how long is it?
        chain_length = i - chain_start
        if chain_length == 1:  # leave it alone
          target[chain_start] = 1
        elif chain_length == 2:  # a chain of two
          if bool(random.getrandbits(1)):
            # do the substitution
            target[chain_start] = -1
            target[i] = 1
          else:
            target[chain_start] = 1
            target[i-1] = 1
        else:  # will get benefit
          target[chain_start] = -1
          target[i] = 1
      chain_start = -1  # not in a chain anymore
    else:  # bit == 1
      if chain_start == -1:
        chain_start = i
  target.reverse()
  return target
```

Listing 2.1: CSD Conversion Algorithm

Figure 2.9 shows the resource utilization when CSD is applied to random matrices of varying element-sparsity. Note that the x-axis is element-sparsity and not bit-sparsity.

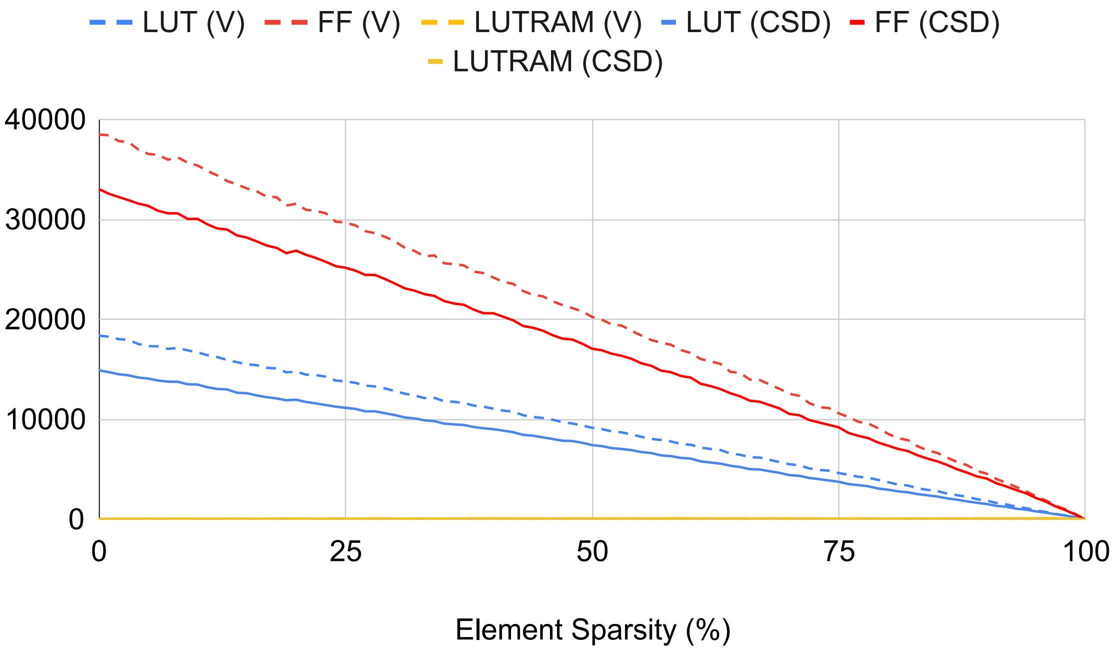


Figure 2.9: CSD Resource Utilization for 64x64 Element-Sparse matrices

CSD results are strictly better than the naive implementation. In this experiment, all the numbers are uniform random. CSD applies equally to them and reduces the hardware by 17% for any level of element-sparsity. In other words, CSD always makes numbers more bit-sparse, which reduces the cost of our architecture. We would expect these savings to improve for larger weight bitwidths.

2.4 Large Scale Design Results

The prior sections looked at the logic synthesis results of small matrices. In this section, we show the scaling of these techniques to much larger matrices. Our experiments in this section use square matrices with dimensions of 512 and 1024, with 8-bit signed weights. We consider element sparsity from 40% to 98%. We use a split matrix to deal with the signed weights, and the split matrices are generated by either splitting positive and negative terms (PN) or by using the CSD technique previously described. The results of these two matrices are fed to an array of final bit-serial subtractors to complete the signed matrix implementation. We ”wrap” the matrix multiplier with a small design that feeds inputs from an SRAM, and captures results in that same SRAM. This design wrapper only adds a few extra LUTs and registers. We run synthesis and place and route on these designs and set a timing constraint of 450MHz. Our target FPGA is the Xilinx XCVU13P [21], which is a 16nm device containing four chiplets in the package (called Super Logic Regions or SLRs). This device has a capacity of 1.7M 6-input LUTs and 3.4M logic flip-flops.

The LUT and register counts as a function of ones in the matrix is plotted in Figure 2.10. The PN and CSD points are based on the identical original signed matrix. The trend lines show the strong correspondence with the resource counts and the number of ones. LUTs are essentially equivalent to the number of ones, and there are two registers per LUT. CSD reduces both the number of ones in the matrix and the resulting resource counts.

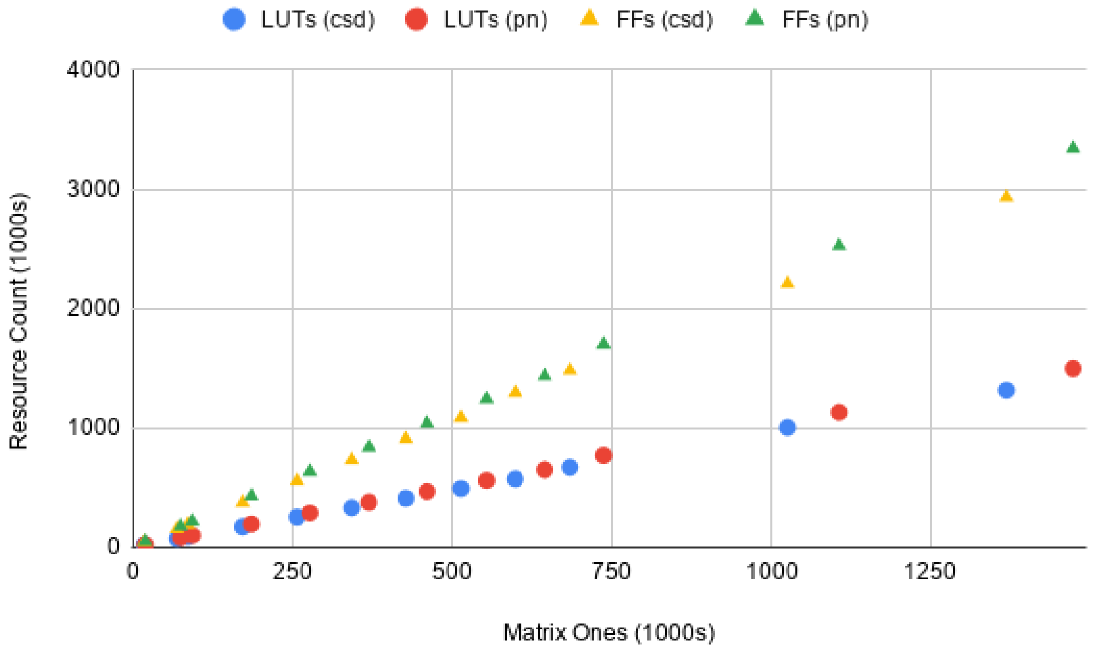


Figure 2.10: Large Scale Area Results: The very strong linear relationship between matrix ones and FPGA resources is obvious.

The process of FPGA design does not always meet the specified timing constraint. Figure 2.11 shows the achieved frequency (Fmax) of these designs after placement and routing. All the paths within these designs have at most one LUT between flops, which means that the frequency is primarily a result of the interconnect delays between LUTs and flops. Other components, such as the SRAMs have a maximum frequency that exceeds 600MHz. There is some noise in these results, but the trends are that bigger matrices run slower. As the matrices get bigger, two mechanisms impact the routing delay:

- The initial layer has a large fanout, approximately corresponding to the dimension times the sparsity. Nets that have a fanout of 100s can have delays of several nanoseconds, which becomes the critical path.
- Nets cross the chiplet boundaries, and those routes have significantly slower propagation delays.

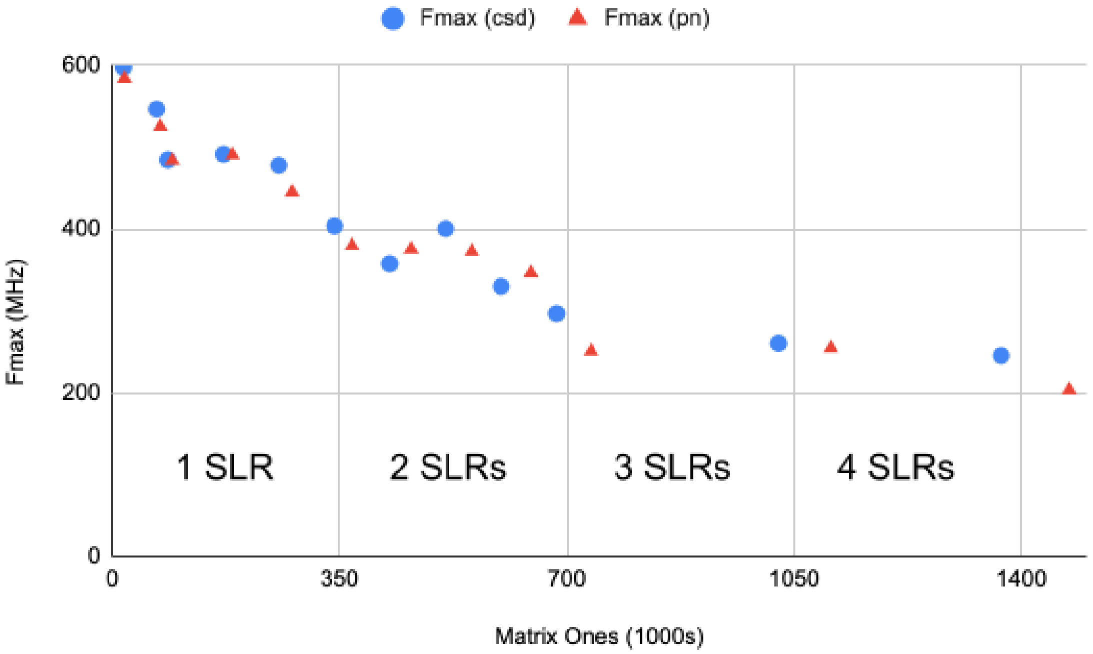


Figure 2.11: Large Scale Frequency Results: Increasing fanout of the first stage and the spanning of multiple chiplets increase the critical path delay and decrease maximum frequency.

Each of the four SLRs within the FPGA have a maximum capacity of 425k LUTs. After about 80% of LUTs are used the tools can struggle. Figure 2.11 has tick marks illustrating the 82% thresholds of SLR capacity. Within one SLR, the frequencies range from 597MHz to 445MHz. Designs requiring 2 SLRs range from 296MHz to 400MHz. Matrices bigger than 2 SLRs seem relatively consistent between 225MHz and 250MHz. Both the fanout and chiplet crossing problems could be addressed by adding registers to perform the fanout and chiplet crossings in multiple cycles. These optimizations are not represented here.

The other limit to the frequency is power. Figure 2.12 plots the estimated total power consumption of this device scaled to run at the maximum achievable frequency. These results were obtained from the Vivado tool based on the default assumptions about switching activity. Under medium settings for airflow and heatsink, the thermal power limit of this FPGA is approximately 150W, which we approach at high dimension and low sparsity.

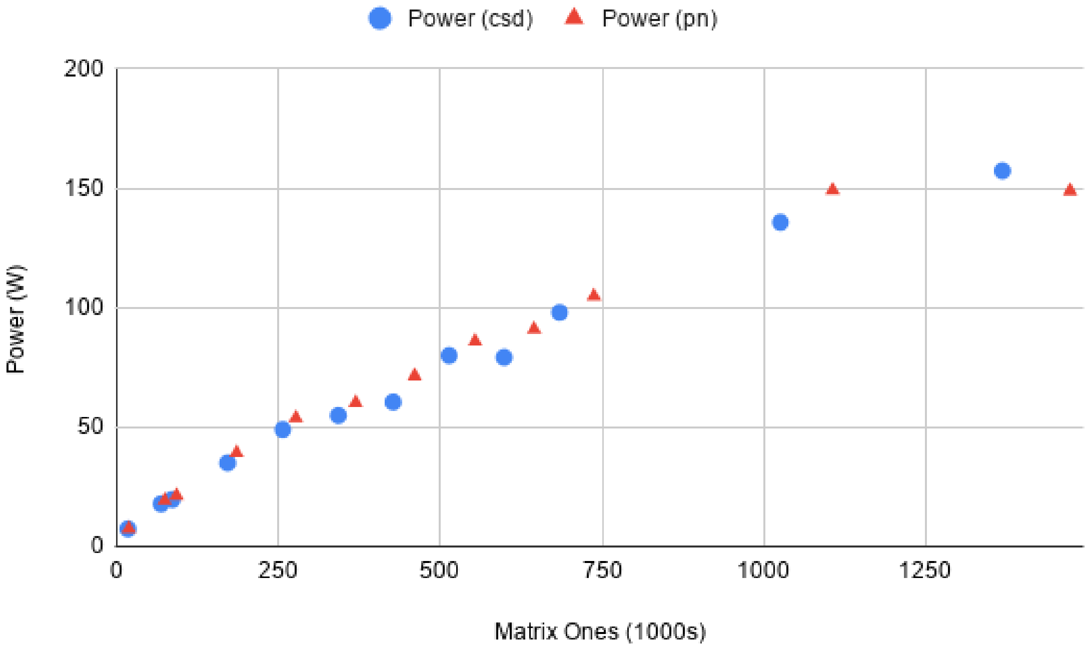


Figure 2.12: Large Scale Power Results: Note the sublinear increase due to the decreasing acheivable frequency. Under medium cooling assumptions, this FPGA has a limit of about 150W.

2.5 Summary

This chapter builds the foundational architecture for our bit-serial matrix multiplier. We show a design which culls away bit-level multiplications by 0. We do so by decomposing the dot-product into a series of boolean dot-products, and then performing constant propagation through the circuit. This allows the hardware utilized to be a function of the number of set bits in the weight matrix. We further reduce hardware utilization using a CSD transform on the weight matrix. We study the hardware utilization, frequency, and power of varying weight matrices on the FPGA. In the next chapter, we evaluate the performance of these implementations against a GPU and SIGMA.

CHAPTER 3

PERFORMANCE EVALUATION

To evaluate our design, we we evaluate the FPGA implementations from Chapter 2 directly against state-of-the-art sparse multiplication libraries on GPU, and also against a recent sparse DNN accelerator.

3.1 Evaluation

3.1.1 GPU Benchmarks

We compare our results against the Volta V100 GPU from NVIDIA. The V100 is built in 12nm technology, whereas the UltraScale+ FPGA is 16nm. The more recent Ampere Architecture from NVIDIA includes hardware support for sparsity up to 50%, but is unfortunately unavailable to us for this study. We use two libraries for the GPU benchmarks: cuSPARSE [25], and a recently published optimized kernel [26]. Neither of these libraries support integer arithmetic, so we are using FP16 as a best-case proxy. We run the FPGA at the maximum frequency achieved after place-and-route, which can be seen in Figure 2.11 and Figure 2.12.

For both of these configurations, we initialize a random weight matrix using the same scheme as described previously. We transfer this to the device and allow the given library to perform any necessary optimizations before timing. Then, we randomly initialize a dense vector, and repeatedly multiply this vector by the matrix so that all caches and memory systems are warm. We do this for 1000 iterations and take the latency to be the mean iteration time. In this case, the latency is measured from the devices memory, through the arithmetic, and back to memory - which is identical to our FPGA implementation.

Sweeping dimension Our first experiment demonstrates latency performance as a function of array dimensions at 98% element sparsity. We sweep the matrix dimension from 64x64 to 4096x4096 in powers of two. The results are show in Figure 3.1 with actual latency numbers and in Figure 3.2 as comparative speedup. The ”Optimized Kernel” is the library in [26].

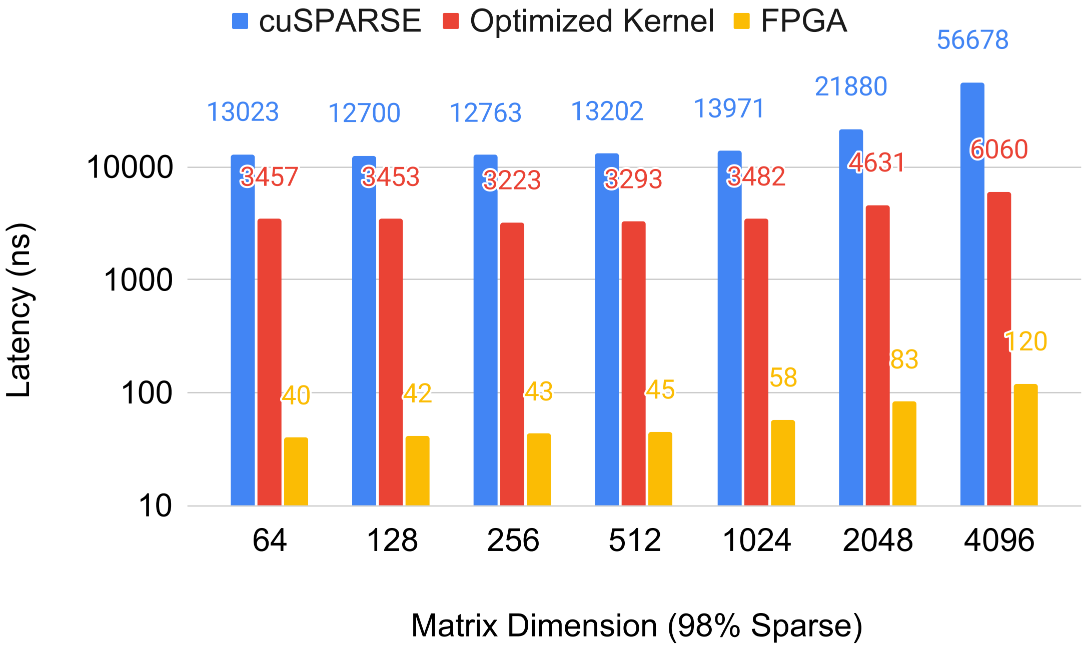


Figure 3.1: Latency (ns) of varying 98% element sparse matrices

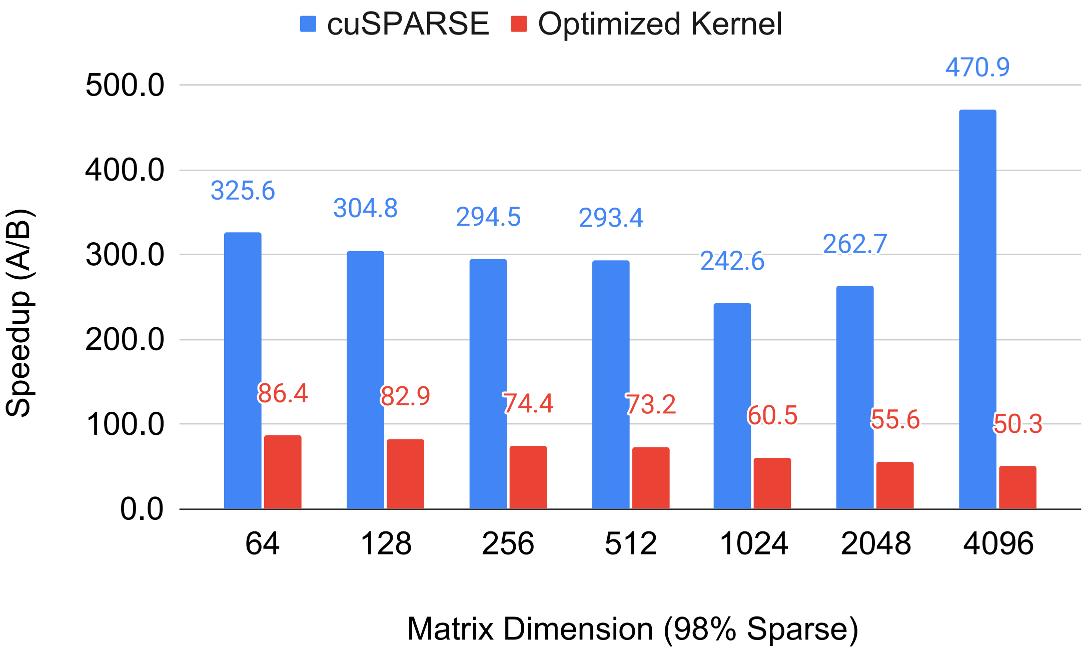


Figure 3.2: Speedup of varying 98% element sparse matrices

Notice that in all cases, our FPGA latency is less than 120ns, whereas the GPU cannot break the 1μs barrier. When the matrix size is less than 512x512, the GPU performance is nearly constant. This indicates the GPU is underutilized and therefore latency bound. GPUs possess incredible throughput using techniques like thread-level parallelism, memory latency hiding, double-buffering, and others. However, all these techniques require the GPU to spawn many more threads than the arithmetic can handle, swapping threads in and out to maximize utilization. In the low-latency regime, these techniques introduce overhead which cannot be overcome, even with the Optimized Kernel. Our solution configures the hardware to exactly support the matrix in question, leading to astounding latency. When the GPU is latency-bound, its latency does not increase for larger matrices. Our solution pays a constant cost to stream the inputs and outputs, but the number of cycles spent reducing partial sum increases logarithmically with matrix size. Furthermore, increasing the matrix size decreases our highest achievable clock frequency. In the GPU's latency-bound regime, we see our speedup fall from 86x to 60x. However, at 1024x1024, the GPU is utilized and is no longer latency-bound, so it begins to see linear scaling with respect to matrix size. Because our solution scales logarithmically in cycles, we see our speedup leveling off at 50x due to the slower clock.

Sweeping sparsity Our next experiment considers the average latency as a function of matrix sparsity at a fixed dimension of 1024x1024. The results are show in Figure 3.3 and Figure 3.4 with average latency and speedup, respectively. Element sparsity is swept from 70% to 98%. Recall that our solution's latency in cycles does not depend on sparsity, but we can clock the design faster as sparsity increases. The GPU sees decreasing latency with increasing sparsity because it needs to operate on fewer elements and the cost of indexing is amortized. For cuSPARSE, increasing sparsity from 70% to 85% sees large reductions in latency as the library reduces the amount of compute to be done. The optimized kernel comparatively spends less time indexing and has higher performance at lower sparsity. In both cases, the compute reduction from 70% to 85% is effective and sees our speedup go from 77x to 72x. As sparsity increases further, the GPU again becomes underutilized and both the latency and speedup level off, yielding a minimum speedup of 60x. Again, the GPU is unable to break the 1μs barrier, whereas our solution stays under 120ns.

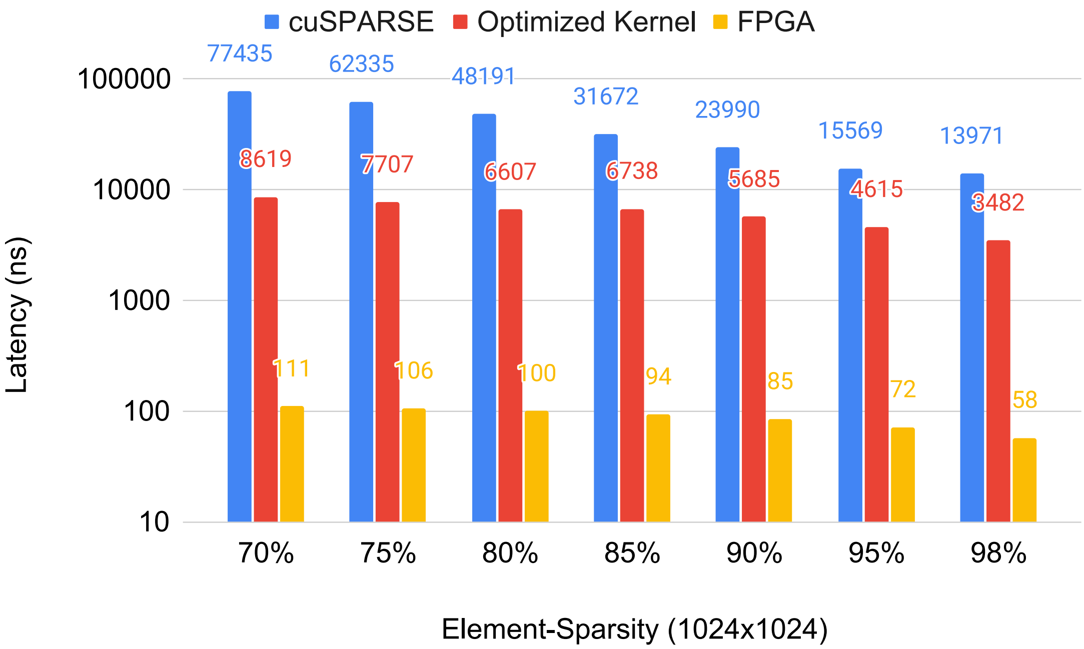


Figure 3.3: Latency of varying sparsity

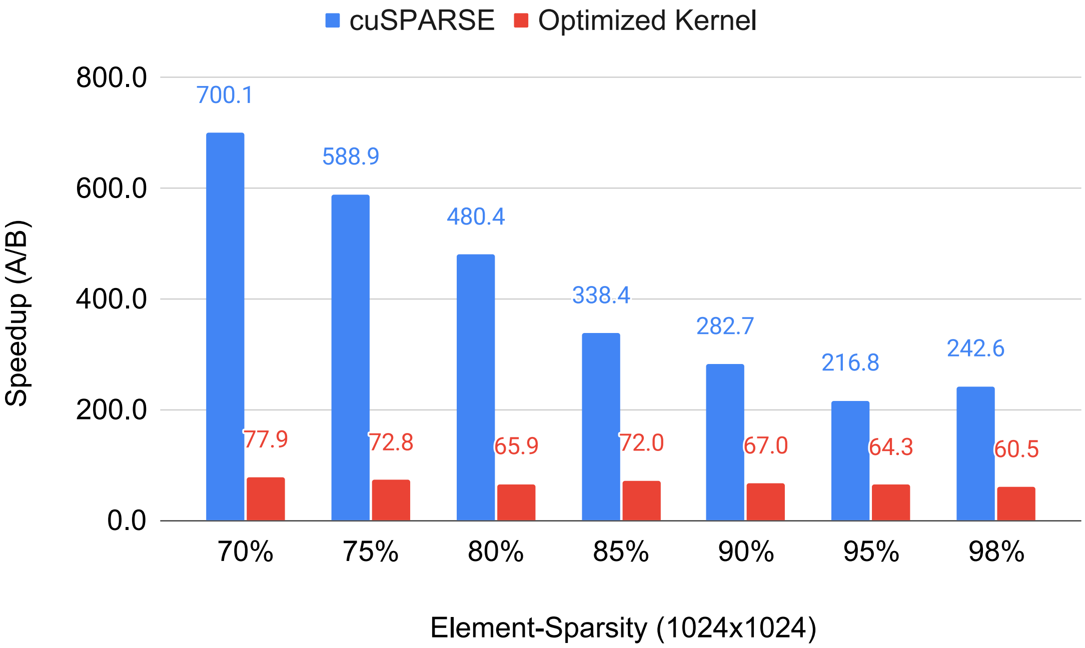


Figure 3.4: Speedup of varying sparsity

Batching We have illustrated the advantage of our architecture for latency-sensitive scenarios, which is our targeted case. In this next experiment, we compare throughput of the GPU solution versus our solution. Previous experiments, because of the recurrence, were done with a batch size of 1, which underutilizes the GPU's pipelined resources. In this experiment, we change the GPU's computation to perform matrix-matrix multiplication, where the number of columns in the multiplicand matrix is the ”batch-size”, borrowing terminology from DNN processing. We again randomly initialize a fixed weight matrix with 95% sparsity and signed 8-bit integers. We choose 95% sparsity to give the GPU ample headroom. We sweep the batch size from 1 to 64, reporting the speedup of our solution compared to the GPU. We are still measuring latency, but in the higher batch sizes, the reciprocal of the latency will approach the maximum throughput of the GPU.

Figure 3.5 presents our results for a 1024x1024 weight matrix, and multiplying a batch_size x 1024 random dense matrix.

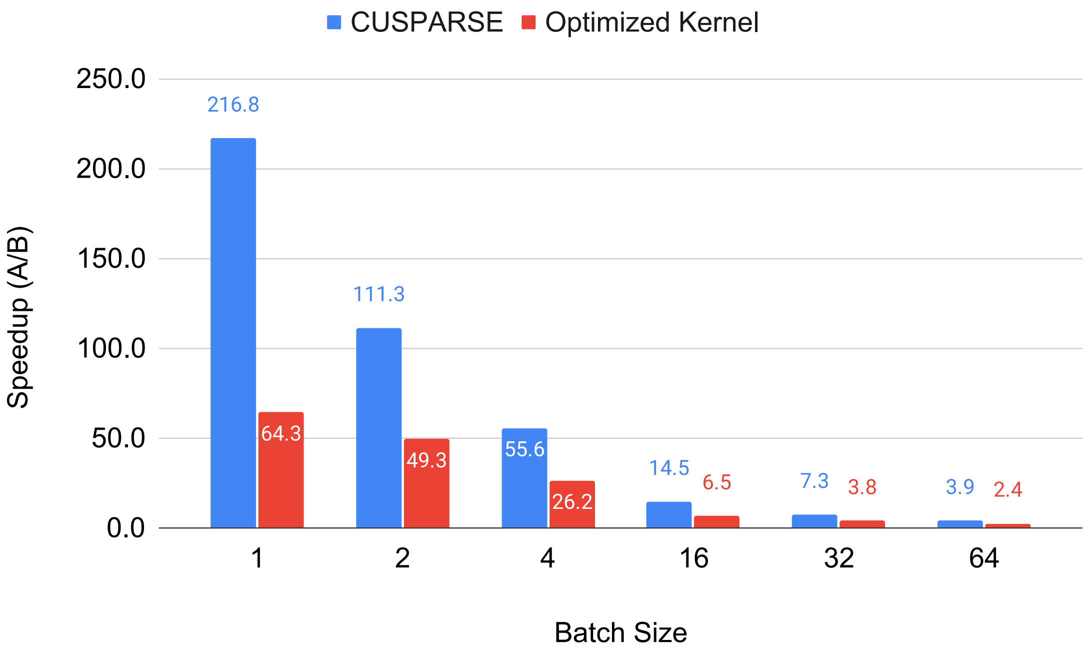


Figure 3.5: Speedup against V100 for varying batch size (matrix = 1024x1024, 95% sparse) Batch-size 1 is a comparison of pure latency. As batch size increases, the speedup is comparing achievable throughput

Figure 3.6 presents our results for a 64x64 weight matrix, and multiplying a batch_size x 64 random dense matrix.

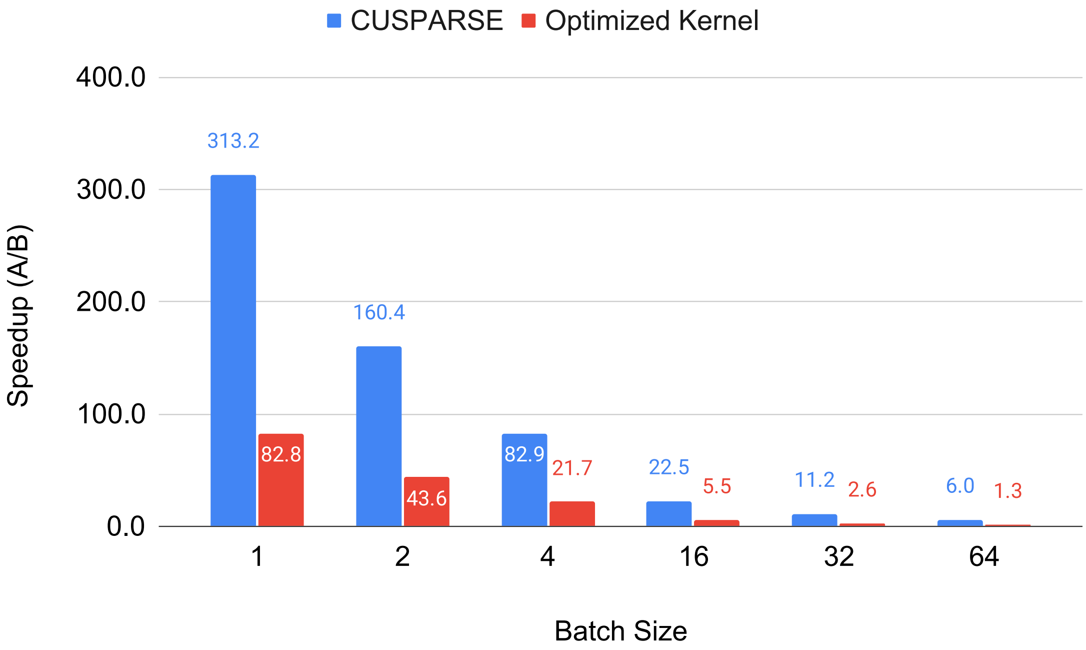


Figure 3.6: Speedup against V100 for varying batch size (matrix = 64x64, 95% sparse) Batch-size 1 is a comparison of pure latency. As batch size increases, the speedup is comparing achievable throughput

As expected, the latency for the GPU solution scales sub-linearly with respect to batch size. As the GPU becomes more utilized, it's able to overlap computation and memory to take full advantage of the massive parallelism. Because our architecture is built for vector products, we have to stream the columns of the input matrix in one-by-one, which yields to linear scaling. Because our solution begins with such a lead, it's able to stream multiple batches before the GPU can reach the same performance. This means our solution is still lower latency at small batch sizes. In the 1024 case, our solution is still marginally better due to the GPU already being close to utilized with the large matrix. With 64x64, the GPU has more computational intensity to fill before it becomes utilized.

The TDP of our FPGA is 150W, while the V100 is 300W, but these numbers are not reliable for accurate run-time comparisons. Given the disparity between the platforms, it is very difficult to accurately compare power consumption between these two solutions. There is a difference in the process technology. The FPGA and GPU also have a different set of peripherals associated with them, that may or may not be active. The dynamic power will be different based on the specific variations of the random weights and activations, which is exacerbated by the int8 vs fp16 difference. Finally, the different boards have different methods to monitor power consumption on their different supply rails. However, we believe that the efficiency gains shown here are due to fundamental computational simplification, and it would be reasonable to assume that the dynamic power would be correspondingly lower.

3.1.2 DNN Accelerator Benchmarks

We also consider the state-of-the-art in Sparse DNN acceleration: SIGMA [15]. SIGMA similarly relies on an input broadcast and reduction tree to perform matrix-multiplication. We reached out to the authors and were given their cycle-accurate simulator. In their paper, they design SIGMA as a 128x128 array of fp16 processing elements (PEs) clocked at 500MHz. To approximate process technology node differences and the change to int8 from fp16, we assume that SIGMA can be clocked at 1GHz. We run SIGMA with the weight matrix stationary and stream the input in to minimize latency.

Sweeping dimension We repeat a similar suite of experiments as before. We sweep the dimension of 98% element-sparse matrices. Figure 3.7 and Figure 3.8 show the latency and speedup respectively.

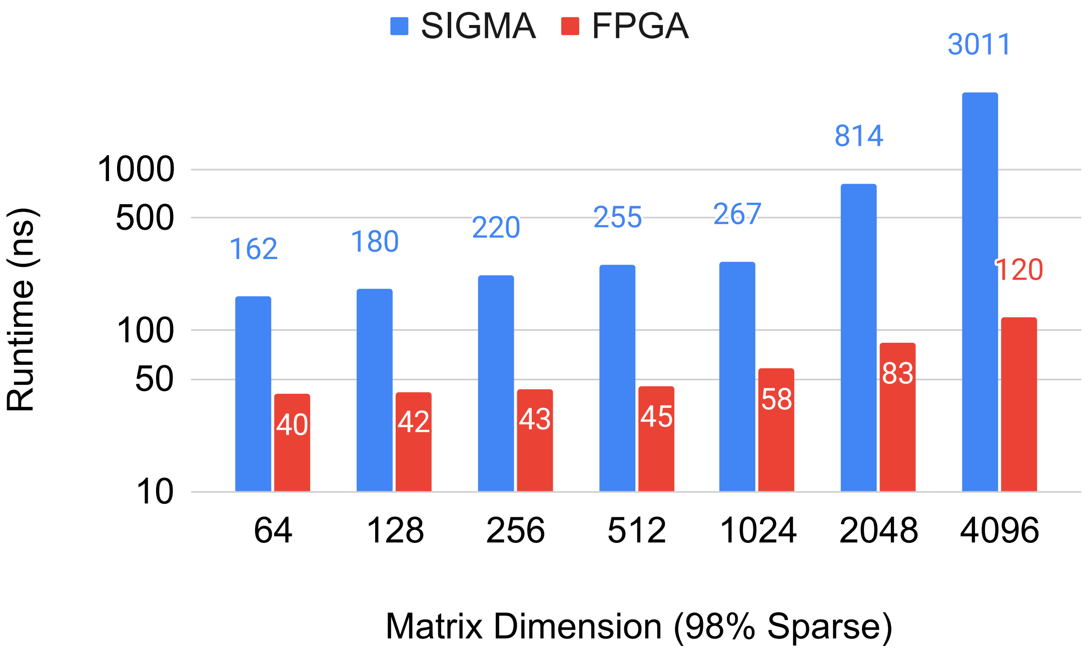


Figure 3.7: Latency of FPGA and SIGMA for varying dimension (98% sparse)

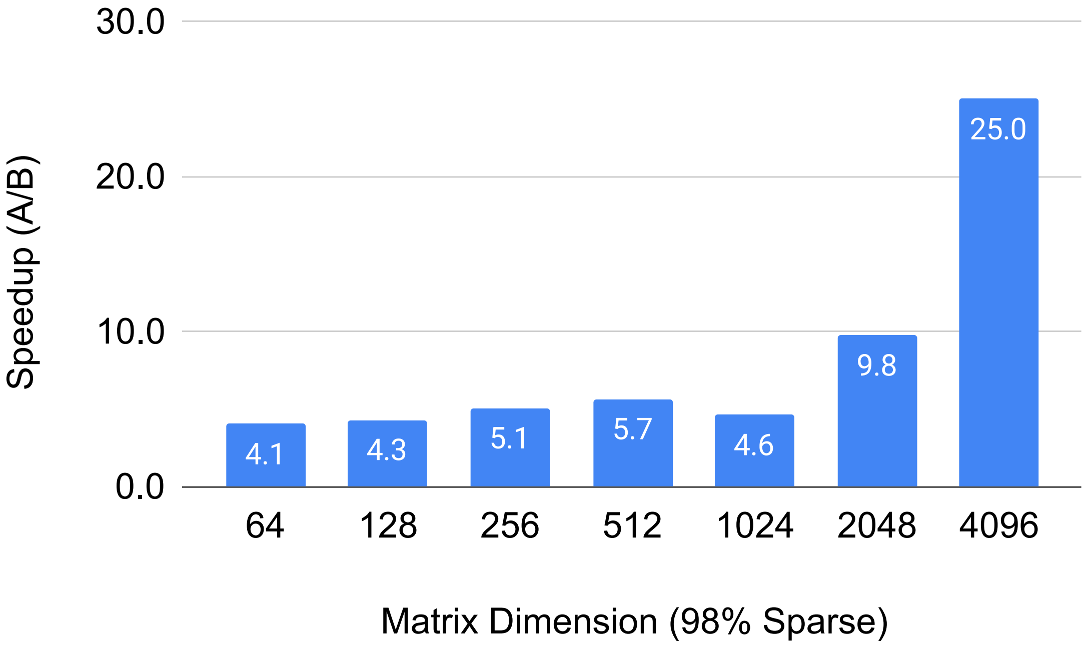


Figure 3.8: Speedup against SIGMA for varying dimension (98% sparse)

For small dimensions, SIGMA does report nanosecond-scale latency due to its input broadcast and reduction tree. Furthermore, small, sparse matrices easily fit into the PE grid, so there is little overhead from tiling. However, after 1024x1024, the elements no longer fit in the PE grid and the computation must be tiled. This invokes extra SRAM use and transitions SIGMA into the memory-bound region, where it sees linear scaling. This yields a 4.1x speedup for our solution in the worst case, but we quickly gain a 25x advantage as the matrix size increases.

Sweeping sparsity Moving on, Figure 3.9 and Figure 3.10 report the latency and speedup of a 1024x1024 matrix, sweeping element-sparsity.

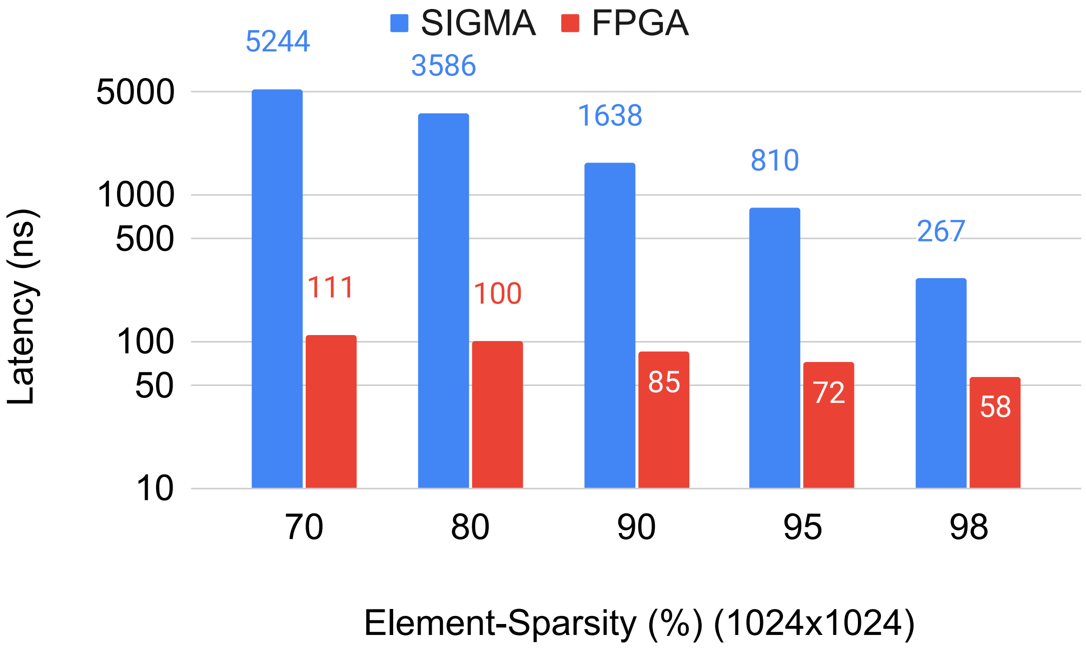


Figure 3.9: Latency of FPGA and SIGMA for varying sparsity (1024x1024)

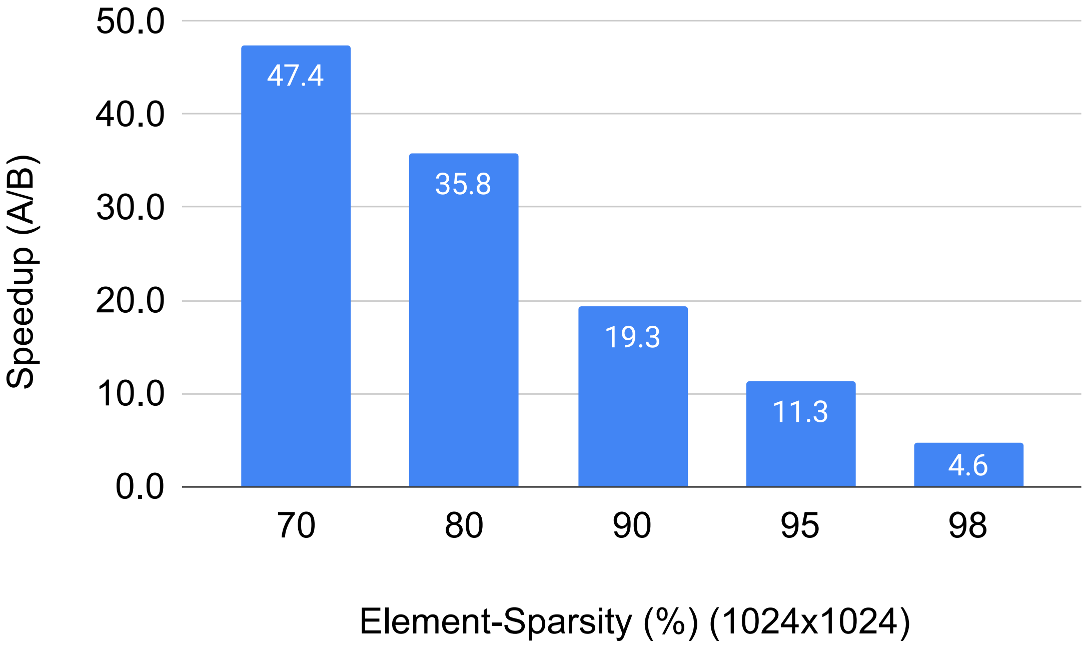


Figure 3.10: Speedup against SIGMA for varying sparsity (1024x1024)

The advantage of SIGMA is that it only maps non-zero weight and activation pairs to PEs. If this union can fit into the PE grid, no tiling is done. As we expect, the size of this set is directly related to the sparsity, and SIGMA sees huge latency improvements as sparsity increases, taking it into the nanosecond regime. However, even 90% sparsity and below is enough to push it back into the microsecond regime, which yields a large advantage to our design.

Batching Finally, we compare matrix-matrix multiplication between our solution and SIGMA. We repeat the same batching test and the result is shown in Figure 3.11.

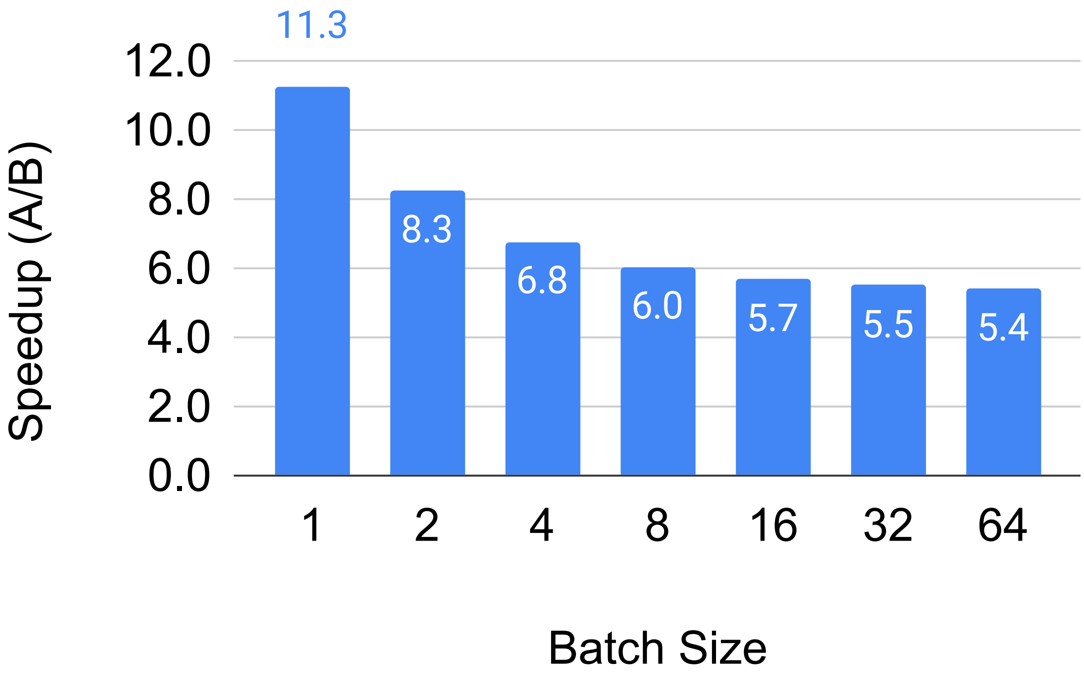


Figure 3.11: Speedup against SIGMA of varying batch-size (1024x1024, 95%) Batch-size 1 is a comparison of pure latency. As batch size increases, the speedup is comparing achievable throughput

Because SIGMA is able to fill its PE array with sparse entries at small batches, it doesn't have as much to gain as a GPU when transitioning from vector to matrix products. At batch-size 2, SIGMA does find opportunity to utilize more PEs and our advantage decreases. However, batch-size 4 and beyond quickly see SIGMA in the memory-bound region again, which causes the speedup to saturate at 5.4x.

3.2 Summary

We evaluate our FPGA design against the state of the art from research and industry. Against V100, our solution is competitive in latency and all sizes and sparsities. This is due to software overhead in the GPU and computational capacity. SIGMA largely solves the overhead due to sparsity detection, but is still limited by hardware capacity. We show our solution is competitive to both in terms of throughput. In the next chapter, we discuss what an ASIC version of this design might look like.

CHAPTER 4

FUTURE WORK:BUILDING AN ASIC

For the given use case, our design has two major limitations: 1) the fanout of the input broadcast saturates the interconnect resources of the FPGA and limits frequency. 2) we are bound by the number of 6-input LUTs in the FPGA, which limits the number of set bits we have in the weight matrix. Creating a CGRA architecture could solve both of these issues. We could employ a broadcast-friendly pipelined interconnect network for the inputs, similar to the Benes network in SIGMA. Furthermore, a 6-input LUT is made using 64 SRAM bits of 6 transistors each, with 64 MUX T-gates of 2 transistors each, which yields a total of 512 transistors for every LUT. A full-adder uses 16 or fewer transistors [27], which is 1/32 the cost. A CGRA implementation of our design would see a grid of full-adders and flip-flops, with a flexible tree-like interconnect to perform partial sums and broadcast interconnect for the input. This approach would allow for higher compute density at higher frequencies.

Even with these optimizations, there may be instances where the compute matrix cannot entirely fit in hardware and must be tiled similar to DNN accelerators. To amortize the cost of loading and moving weights, DNN inference accelerators often employ batching to perform the multiplication of multiple vectors with the same matrix, which saves power by reducing the movement of matrix weights. Our approach eliminates the movement of matrix weights by programming them into an interconnect matrix and takes further advantage by propagating the constants within the matrix. The time to modify the interconnect matrix of the FPGA is on the order of 200ms, which limits its practicality in moving weights during runtime.

However, the feed-forward topology of this network allows for the approach of pipeline reconfiguration [28]. Pipeline configuration is the ability to configure hardware cycle-by-cycle as the pipeline fills. This concept would allow a reconfigurable platform with nearly zero configuration overhead time to change the matrix weights. Pipeline configuration is not supported in conventional FPGAs, but we could support it in a custom CGRA architecture. As the partial sums travel down the tree, the levels above the current accumulation can be reconfigured as their state is no longer needed. One can think of ”waves” of configuration travelling down the tree, where some parts of the tree are reconfiguring and some parts are computing. This is analogous to double-buffering in DNN processing.

We plan to explore these optimizations in future work.

CHAPTER 5

CONCLUSION

We present an architecture for performing integer matrix-vector multiplication. The cost and power of our implementations scale exactly with the number of non-zero terms in the matrix. Bit serial implementations allow for us to support matrices with up to 1.5 million ones, as large as 1024x1024 eight-bit matrix at a sparsity of 60%. These products can be produced with nanosecond-scale latencies, which is 2-3 orders of magnitude faster than a GPU, and 4-47x faster than a sparse accelerator. Throughput is also competitive, especially for large matrices. Given an FPGA's slow configuration time, these techniques are primarily suitable to computations where the matrix is essentially fixed, and where low latency is critical, such as reservoir computing. However, a customized programmable device for this approach could pipeline the configuration, thus effectively hiding it, and enable this approach to work for dynamic sparse matrices.

REFERENCES

[1] M. Tan and Q. V. Le, *Efficientnet: Rethinking model scaling for convolutional neural networks*, 2020. arXiv: 1905.11946 `[cs.LG]`.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, *Attention is all you need*, 2017. arXiv: 1706.03762 `[cs.CL]`.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, jan 2016.

[4] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "Cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. arXiv: 1410.0759.

[5] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2016. arXiv: 1510.00149 `[cs.CV]`.

[6] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *CoRR*, vol. abs/1704.04760, 2017. arXiv: 1704.04760.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012, pp. 1097–1105.

[8] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Ł. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and

J. Dean, *Google's neural machine translation system: Bridging the gap between human and machine translation*, 2016. arXiv: 1609.08144 `[cs.CL]`.

[9] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, Salt Lake City, Utah, USA: Association for Computing Machinery, 2014, pp. 269–284, ISBN: 9781450323055.

[10] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, *Dissecting the nvidia volta gpu architecture via microbenchmarking*, 2018. arXiv: 1804.06826 `[cs.DC]`.

[11] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, *Understanding reuse, performance, and hardware cost of dnn dataflows: A data-centric approach using maestro*, 2020. arXiv: 1805.02566 `[cs.DC]`.

[12] S. Sharify, A. D. Lascorz, M. Mahmoud, M. Nikolic, K. Siu, D. M. Stuart, Z. Poulos, and A. Moshovos, "Laconic deep learning inference acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 304–317, ISBN: 9781450366694.

[13] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[14] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, *Eie: Efficient inference engine on compressed deep neural network*, 2016. arXiv: 1602.01528 `[cs.CV]`.

[15] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.

[16] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *SIGPLAN Not.*, vol. 53, no. 2, pp. 461–475, Mar. 2018.

[17] A. F. Murray, A. V. W. Smith, and Z. F. Butler, "Bit-serial neural networks," in *Neural Information Processing Systems*, D. Z. Anderson, Ed., American Institute of Physics, 1988, pp. 573–583.

[18] P. Judd, J. Albericio, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 80–83, 2017.

[19] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, “Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks,” *CoRR*, vol. abs/1712.01507, 2017. arXiv: 1712.01507.

[20] J. Albericio, P. Judd, A. D. Lascorz, S. Sharify, and A. Moshovos, “Bit-pragmatic deep neural network computing,” *CoRR*, vol. abs/1610.06920, 2016. arXiv: 1610.06920.

[21] Xilinx, *Virtex ultrascale+*, https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html, 2020 (accessed Sept 4, 2020).

[22] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” 2016. arXiv: 1510.00149 `[cs.CV]`.

[23] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, *Eie: Efficient inference engine on compressed deep neural network*, 2016. arXiv: 1602.01528 `[cs.CV]`.

[24] A. Avizienis, “Signed-digit number representations for fast parallel arithmetic,” *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 389–400, 1961.

[25] NVIDIA, *Nvidia cusparse api reference*, https://docs.nvidia.com/cuda/cusparse/index.html, 2020 (accessed Sept 4, 2020).

[26] T. Gale, M. Zaharia, C. Young, and E. Elsen, *Sparse gpu kernels for deep learning*, 2020. arXiv: 2006.10901 `[cs.LG]`.

[27] A. Dubey, S. Akashe, and S. Dubey, “A novel high-performance cmos 1 bit full-adder cell,” in *2013 7th International Conference on Intelligent Systems and Control (ISCO)*, 2013, pp. 312–315.

[28] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, “Piperench: A co/processor for streaming multimedia acceleration,” in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ser. ISCA ’99, Atlanta, Georgia, USA: IEEE Computer Society, 1999, pp. 28–39, ISBN: 0769501702.