Efficient Hardware Realization of Convolutional Neural Networks using Intra-Kernel Regular Pruning

by

Maurice Muyun Yang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Convolutional neural networks (CNNs) have proven their success in a wide range of applications. While CNNs boast remarkable performance, they require significant computational and memory resources for operation. As research strives towards higher classification accuracy, CNN topologies have increased in depth, complexity and size. In response, algorithmic-level optimizations have been proposed to reduce the size of CNNs while retaining classification accuracy. While these advances promise savings in theory, they often underperform in practice, especially when adopted into hardware. In order achieve practical savings, algorithmic changes must be considered in the perspective of hardware, thus necessitating a software-hardware codesign philosophy.

We propose an Intra-Kernel Regular (IKR) pruning scheme to reduce the size and computational complexity of CNNs by removing redundant weights at a fine-grained level without loss in classification accuracy. Unlike other pruning methods such as Fine-Grained pruning, IKR pruning maintains regular kernel structures and employs data compression techniques that translate well into hardware. At the hardware level, we propose an FPGA-design framework targeting IKR-pruned CNNs. The organisational structure of the design enables potential for high parallelism and efficient utilization of on-chip resources. Experimental results in software demonstrate up to 10× reduction in weights and 7× reduction in computation at a cost of less than 1% degradation in accuracy versus the un-pruned case. Evaluation of the accelerator show computational speeds up to 77.7 GOP/S (effectively 403 GOP/S) with each DSP effectively performing 0.53 GOP/S.

Acknowledgements

First and foremost, I would like to thank my advisor Vincent Gaudet for his guidance, equipment and inspiration that he gave me in my graduate studies. Thanks to him, I had the chance to do research with talented students in an exciting field.

Second, I would like to thank the graduate students who contributed to my research. Most notably, Assem Hussein and Mahmoud Faraj have had a positive impact in my research.

Lastly, I would like to thank the members of my thesis committee, Professor Mark Aagaard and Professor Fakhri Karray. They have helped me improve the final version of this document.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent advances in computing technologies and data acquisition have paved the way for deep learning, with convolutional neural networks (CNNs) emerging as a dominant computing architecture for applications involving image recognition and object detection [11]. Consequently, CNNs have been deployed in many avenues of society. Their success has enabled our devices to be perceptive of their surroundings, opening the avenue to an enormous number of new possibilities.

Although CNNs are proficient in tasks such as image classification (proven in classification challenges such as MNIST [32], CIFAR-10 [27] and ImageNet [15]), operating these networks require high computational and memory resources. As advances strive towards higher classification accuracy, CNN topologies tend to be deeper and more complex. Modern CNNs often require the training of millions of weight parameters and require hundreds of millions arithmetic operations for a single inference [28]. While the quantity of calculations appear daunting, the majority of them are highly parallelisable. Accelerators often exploit this property to operate CNNs at high speeds.

Neural network acceleration has been explored on both general-purpose processors (graphical processing units (GPUs) and central processing units (CPUs)) and domain specific hardware (field programmable gate arrays (FPGAs) and application specific integrated circuits (ASICs)). Generally, CPUs are specialized for serial operation, making

them poor candidates for parallel computation. GPUs offer massively parallel processing power and are well suited to perform CNN calculations. However, GPUs also suffer from high power consumption and a large form factor, which limits their viability in certain applications. FPGAs and ASICs offer highly customizable and parallel designs; but are constrained by potentially long and tedious design cycles. The choice of platform presents a trade-off between performance and flexibility versus development time. FPGAs embody the flexibility of hardware while having development time comparable to general purpose processors, making them good candidates for hardware exploration and prototyping. This thesis will focus on CNN acceleration using FPGAs.

1.1 FPGA for Neural Networks

FPGAs are used a wide range of data processing applications. Their capability for massive parallelization and their versatility in customization and re-programmability make FPGAs an attractive option. Thematically, they reside between general-purpose processors, which uses software to direct general-purpose hardware, and ASICs, which are customized hardware specific for an application. FPGAs comprise thousands of generalized hardware blocks that can be configured and interconnected to embody customized hardware. This platform offers the ability to build custom pipelined datapaths, enabling the parallel execution of operations that may burden traditional software.

CNNs operation is complex and resource intensive, making their incorporation in mobile applications a challenge. While cloud-based deep learning offer a solution by relieving the computational burden from the device to GPU farms in the cloud, the communication latency is an issue for time-critical tasks involving real-time operations. Consequently, there is keen research interest in custom neural network hardware for edge computation; which is the philosophy of placing processing power directly the edge of the application where data is generated. Past research have presented implementations of custom CNN accelerators using FPGAs [11] [44] [49].

1.2 Sparse Convolutional Neural Networks

Weights in CNN models vary vastly in importance. Studies have proven that large portions of weights can be removed, or pruned, without detriment to classification accuracy. The work of [21] demonstrated a 90% reduction of weights with negligible accuracy degradation (less than 1%). Theoretically, removing weights is extremely advantageous, since less memory is required to store the network model and less computation is required for each inference operation. In practice, however, pruning produces irregular network connections that translates poorly in a hardware design, making it difficult to attain practical savings. This thesis explores a novel pruning algorithm that aims to produce structured network connections suitable for hardware acceleration.

1.3 Report Structure

This thesis contains the following chapters:

Chapter 2: Background gives an overview of the mathematical model of CNNs and outlines target benchmarks and network architectures used for experimentation.

Chapter 3: CNN Acceleration and Optimization gives an overview of state of the art CNN accelerators and describes common strategies used in CNN acceleration.

Chapter 4: Sparse Convolutional Neural Network Acceleration details our novel approach to pruning at the algorithmic level and presents the system architecture of our proposed accelerator.

Chapter 5: Experimental Results compares the compression rate and the hardware performance of our work to other relevant implementations in literature.

Chapter 6: Conclusions provides concluding remarks and presents possible additions further research opportunities.

The main contribution of the thesis is the formulation of a novel pruning scheme that uses generated pruning patterns to preserve important weights while eliminating unimportant weights at the intra-kernel level. Our approach reaps the benefits from fine-grained pruning while maintaining predictable kernel patterns that can be exploited using specialized hardware. Moreover, the resulting sparse kernels can be stored very compactly in compressed sparse pattern (CSP) format, a representation that exclusively keeps non-zero weights and the corresponding mask index. To validate our scheme, we designed a hardware accelerator tailored to operate the resulting sparse CNN.

Chapter 2

Background

This section gives a primer on the mathematical model behind CNNs. Section 2.1 introduces the idea of an artificial neural network, Section 2.2 describes the operation of CNNs, including all the layers, and Section 2.3 details the CNNs targeted for pruning and acceleration.

2.1 Artificial Neural Networks

An artificial neural network (ANN) [37] is a computer model that excels in prediction and pattern recognition. Biologically inspired, an ANN mimics after the structure of the human brain, inheriting its plasticity. An ANN is not meticulously handcrafted, but rather only its overarching structure is defined. Through *training* [42], the ANN gradually improves at performing a task by incrementally modifying the network model. The flexibility of ANNs allows their use in a large spectrum of applications, given that sufficient training data is available. Supervised training involves labeled data, whereas unsupervised training does not. While the concept of the ANN was introduced in the 1950s, there has been a resurgence in interest in recent years due to the accessibility of a large amount of data and advancements in computation.

Neural networks are comprised of interconnected artificial neurons organized into groups

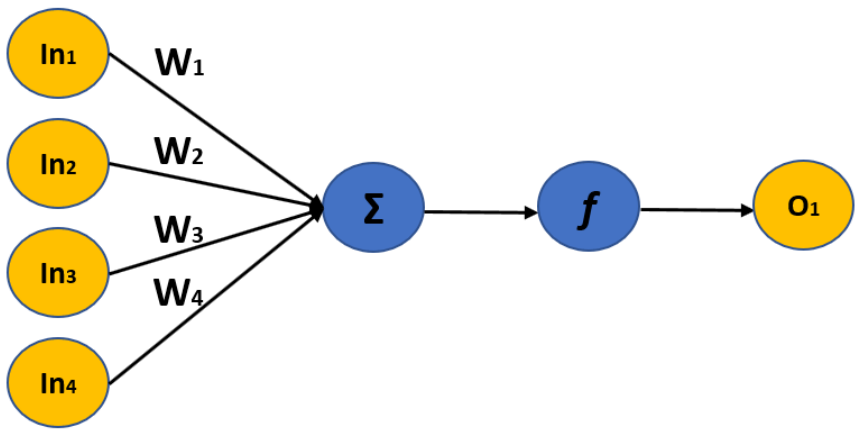


Figure 2.1: Depiction of a single neuron with connections to four input neurons.

called layers. These layers are cascaded sequentially, with neurons within each layer responding to the activity of neurons in the previous layer. Connections between neurons have an associated weight that describes the strength of the neural connection, or the sensitivity of the neuron to others activity. These weights are tuned during training such that neurons are sensitive to particular patterns of neural activity. Fig. 2.1 shows a depiction of a neuron with four input connections. When data is fed through the neural network, neurons in the first layer will activate in response to patterns or features in the input. In successive layers, neurons will activate due to patterns observed in previous layers, effectively extracting patterns from patterns. Neural networks are defined by the organization and the composition of its layers; categories of neural networks have spawned from permutations of these traits. The most straightforward type is the multilayer perceptron (MLP), which follows a feedforward structure, where inputs to a layers neurons stems from neurons from the previous layer. The convolutional neural network (CNN) is an extension upon the MLP by incorporating convolutional and pooling layers.

2.2 Convolutional Neural Networks

A Convolutional Neural Network (CNN) [31] is an extension of the Artificial Neural Network and specializes in object recognition. This topology resolves the issues [31] surrounding the classical ANN model by introducing the convolutional layer and the pooling (subsampling) layer. These new layers are reorganisations of the neural connection patterns found in the fully connected layer, such that these layers are spatially aware. The convolutional layer can detect features in 2D space, of which can then be merged further into high level features.

2.2.1 Layer Types

Convolutional Layer

The role of a convolutional layer is to locate features in input images, where features of interest are represented by collections of weights called kernels [18]. To find features within an image, kernels are convolved with the input image. By scanning the image for features, a record of located features called feature maps is built. Further convolution on feature maps allows the recognition of higher order features, giving better insight on the nature of the image.

The convolutional layer performs feature extraction by performing the convolution operation on the input images against a set of kernels. A kernel is a 2D matrix containing numbers called weights, where the values and the spatial locality of the weights depict the feature. The convolutional layer receives N input feature maps and produces M output feature maps. Connections between input and output are represented by M filters, each of which has N kernels of dimensions $K \times K$. These parameters are summarized in Table. 2.1. The convolutional layer performs convolutions between input feature maps and kernels to generate output feature maps, as shown in (2.1), where f_i^{out} denotes the i-th output feature map, f_j^{in} denotes the j-th input feature map, $k_{i,j}$ denotes the j-th kernel in the i-th filter and b_i denotes the i-th bias. It can be seen from Pseudocode 1 that the convolutional layer is comprised of by multiply and accumulate operations (MAC).

The computational and space complexity for the convolutional layer are summarized in (2.2) and (2.3). It is observed that the computational complexity is $R \times C$ times higher than the space complexity, indicating that the computation to memory access ratio is large.

$$f_i^{out} = \sum_i^M \sum_j^N \left(f_j^{in} * k_{i,j} + b_i \right) \tag{2.1}$$

$$runtime \sim \mathcal{O}(N * M * R * C * K^2) \tag{2.2}$$

$$space \sim \mathcal{O}(N * M * K^2) \tag{2.3}$$

Pseudocode 1

```
for(n=0; n<N; ++n):
   for(m=0; m<M; ++m):
      for(r=0; r<R; ++r):
         for(c=0; c<C; ++c):
             for(x=0; x<K; ++x):
                for(y=0; y<K; ++y):
                   output_fm[m][r][c] +=
                      kernel[n][m][x][y] *
                      input_fm[n][x+r][y+c]
```

Fully Connected Layer

A fully conected (FC) layer contains all to all connections between the input and output neurons and weights in this layer are grouped into one 2-D matrix. Output neurons are calculated through a vector-matrix multiplication between the vector of inputs and the weight matrix. The layer receives N neurons as inputs, produces M neurons as outputs

Table 2.1: Description of loop parameters for the convolutional layer.

Parameter	Description
N	Number of input feature maps
M	Number of output feature maps
R / C	Number of rows / columns of an input feature map
K	The height / width of each kernel

and comprises $N \times M$ weights. This operation is summed up in (2.4), where X^{in} denotes the vector of input neurons, Y^{out} denotes the vector of output neurons, W denotes the weight matrix and b denotes the vector of biases.

$$Y^{out} = W \cdot X^{in} + b \tag{2.4}$$

$$runtime \sim \mathcal{O}(N * M) \tag{2.5}$$

$$space \sim \mathcal{O}(N * M) \tag{2.6}$$

The computational and space complexity for the fully connected layer are summarized in (2.5) and (2.6). Notably, the computational complexity and space complexity are the same, indicating that the computation to memory access ratio is one.

Rectified Linear Unit (ReLU) Activation Layer

Activation layers are incorporated into the CNN to introduce non-linearity. While older CNN models often employ the activation functions of tanh or sigmoid, these functions suffer from the vanishing gradient problem [38], an issue that hinders the training of CNNs. As a result, modern CNNs have adopted the ReLU [38] activation function. Shown in (2.7), ReLU sets the output as the maximum between the input and zero.

$$f(x) = max(x, 0) \tag{2.7}$$

Pooling Layer

A pooling layer [31] is usually inserted between successive convolution layers to reduce the number of parameters and lowers the required amount of computation in the network. It partitions the input feature maps into submatrices and then subsamples each submatrix into one value. Max pooling is popularly used, in which the output is equal to the maximum value in the submatrix.

2.2.2 Training

CNNs are adapted towards the desired behavior through training. A large training set is required to train a CNN, often comprising of thousands of images and corresponding labels. Training is conducted in two phases. In the first phase, images from the training set are fed into the CNN in feedforward configuration and the network outputs a prediction on the nature of the image. To gauge the prediction error, a loss function l is used to calculate the difference between the prediction and the actual label of the image. A large loss indicates that the prediction is erroneous, while a loss of zero indicates a perfect prediction. The second phase involves correcting the network model through backpropagation using gradient descent [41], where each weight is individually adjusted based on its impact on the prediction result and the loss. Weights are updated according to (2.8), where α denotes the learning rate.

$$w^{new} = w^{old} - \alpha \frac{\partial l}{\partial w}(w^{old}) \tag{2.8}$$

Table 2.2: Architecture of LeNet-5 and VGG-8. 2x128C3 denotes two adjacent convolutional layers having 128 feature maps each and kernels of dimensions 3 × 3. MP2 denotes one non-overlapping max pooling layer with dimensions 2 × 2 and stride 2. 10FC denotes an FC layer with 10 output nodes.

Network	Architecture	DataSet
LeNet-5	1x20C5-MP2-1x50C5-MP2-500FC-10FC	MNIST
VGG-8	2x128C3-MP2-2x128C3-MP2-2x256C3-256FC-10FC	CIFAR-10

Table 2.3: Overview of Parameters and Computational Requirements of LeNet-5, including number of input and output neurons, number of weights and required number of operations in millions.

Layer	Notation	# Inputs	# Outputs	# Weights	Required MOPS
1	1x20C5	784	11520	500	0.58
2	1x50C5	2880	3200	25000	3.2
3	500FC	800	500	400000	0.8
4	10FC	500	10	5000	0.01
Total		4404	15230	430500	4.59

2.3 Target Neural Networks and Datasets

We conducted our study targetting two CNNs, LeNet-5, introduced in [30], and a custom CNN inspired by VGG [43] denoted as VGG-8. Each CNN targets a benchmark dataset, namely MNIST [32] for LeNet-5 and CIFAR-10 [27] for VGG-8. MNIST is a collection of 28 x 28 greyscale images, with each image containing a single handwritten digit from 0 to 9. CIFAR-10 is a collection of 32 x 32 RGB images, with each image belonging to one of 10 object classes, such as cat, frog, airplane, etc. Since these benchmarks are standard and these CNNs resemble other CNN topologies found in research, this setup offer a good avenue for the comparison of our results against literature.

The architectures of the two networks are outlined in Table 2.2. To describe the architecture of CNNs, the following notation is adopted:

Table 2.4: Overview of Parameters and Computational Requirements of VGG-8, including number of input and output neurons, number of weights and required number of operations in millions.

Layer	Notation	# Inputs	# Outputs	# Weights	Required MOPS
1	1x128C3	3072	115200	3456	18.66
2	1x128C3	115200	100352	147456	693.6
3	1x128C3	25088	18432	147456	127.4
4	1x128C3	18432	12800	147456	88.47
5	1x256C3	6400	2304	589824	31.85
6	1x256C3	2304	256	589824	3.539
7	256FC	256	500	65536	0.131
8	10FC	256	10	2560	0.00512
Total		171008	249610	1693568	968.7

- Convolutional layer: 2x128C3 denotes two adjacent convolutional layers having 128 feature maps each and kernels of dimensions 3 x 3.
- Pooling layer: MP2 denotes one non-overlapping max pooling layer with dimensions 2 x 2 and stride 2. 2. Dropout [25] is applied after each MP2 layer with 50% keep probability [25] to prevent overfitting.
- Fully connected layer: 10FC denotes an FC layer with 10 output nodes.

Table 2.3 and Table 2.4 outline the layer by layer statistics for the number of inputs, outputs and weights and the computational requirements in millions of multiply or addition operations (MOP). Evidently, the convolutional layers are the most demanding in terms of operations.

Chapter 3

CNN Acceleration and Optimization

This section gives an overview of the strategies and optimizations for CNN acceleration in hardware. Section 3.1 outlines the state of research in CNN acceleration and strategies that are relevant in our implementation, Section 3.2 describes the concept of pruning and discusses previous pruning schemes found in literature, and Section 3.3 defines the specifications of the hardware platform used in our design.

3.1 Dense CNN Acceleration

Hardware acceleration of CNNs has been widely researched in the past. Early works explored parallelism in CNN computation; leveraging the data-level parallelism to build high performance compute engines. The works of [23], [39] [49], observed that for large CNNs, the size of the network model often exceeds the capacity of on-chip memory, which necessitates its storage off-chip. To reduce the volume of external data transfer, these groups evaluated various strategies, including data reuse, caching of intermediate results and data quantization. With quantization, data precision can often be reduced down to 16-bit fixed point without loss in network accuracy, as shown in [11], [34]. Advanced techniques such as weight sharing [20] and dynamic precision [39] can reduce precision further down to 8/4 bits for the convolutional/FC layer respectively. For designs limited

by on-chip memory capacity, loop tiling [39], [49] can be used to reschedule the workload to buffer data more effectively.

3.1.1 Computational Workload

Parallelization refers to the replication hardware resources to compute multiple operations simultaneously. Due to the enormous computational complexity of CNNs, parallelization is crucial for high performance acceleration. However, it should be noted that since multiplications and additions comprise the majority of operations, hardware replication will often consume DSPs [3], which are limited in supply. For inference, operations are independent within each layer, giving us a high degree of strategical flexibility. By analysing Pseudocode 2, it is seen that parallelization translates to unrolling certain loops. For the convolution layer we consider three degrees of parallelism [39]:

- Inter-kernel parallelism considers the two innermost loops involved in calculating the elementwise multiply and accumulate (MAC) operation between a kernel and a windowed selection of an input feature map.
- Intra-output parallelism considers the loop involving N. It considers the simultaneous convolution of multiple input feature maps in computing a single output feature map.
- Inter-output parallelism considers the loop involving M. It considers the simultaneous convolution involving a single input feature map in computing multiple output feature maps.

Due to the computationally intensive nature of CNN inference, a fully parallel approach is impractical. For example, fully parallelizing the convolution layer, translates to unrolling all the loops in Pseudocode 2. Considering a small convolution layer with a single 32×32 sized input feature map, 5×5 sized kernels and twenty 28×28 sized output feature maps, a fully parallel design must accomodate 0.58 million simutaneous multiplies/adds, which requires an impractical amount of resources. Thus loops should be selectively unrolled in accordance to a practical design.

Loop tiling is an optimization that partitions loop iterations into smaller blocks. In computer science it is often used in linear algebraic algorithms to reduce data access latency, by ensuring that data used in the loop stays in the cache for reuse. As an example, Pseudocode 3 depicts a normal vector-matrix multiplication and a tiled vector-matrix multiplication algorithm. In the normal algorithm, if the cache cannot encompass the entire x vector, then data must be constantly fetched from memory and no data will be reused; resulting in cache misses. In the tiled algorithm, if the tile factor B is chosen such that the B sized slice of x vector is small enough to fit in the cache, then vector will remain in cache for reuse, resulting in cache hits. The concept of loop tiling can be incorporated in CNN acceleration, as an extension to the parallelism strategies. If an entire loop cannot be fully unrolled, it can be instead partially unrolled using tiling; where the tiling factor represents the number of simultaneous operations.

Pseudocode 2

```
//unroll for intra_output parallelism
for(n=0; n<N; ++n):
//unroll for inter_output parallelism
   for(m=0; m<M; ++m):

      for(r=0; r<R; ++r):
         for(c=0; c<C; ++c):

//unroll for inter_kernel parallelism
            for(x=0; x<K; ++x):
               for(y=0; y<K; ++y):
                  output_fm[m][r][c] +=
                     kernel[n][m][x][y] *
                     input_fm[n][x+r][y+c]
```

3.1.2 Algorithmic Optimizations

In order to accelerate both the convolution and fully connected layers using the same hardware, operations for each layer are translated into general matrix multiplications (GEMMs). This transformation maps kernels and input feature maps / neurons into matrices, and allows us to partition the overall complicated problem into straightforward vector operations.

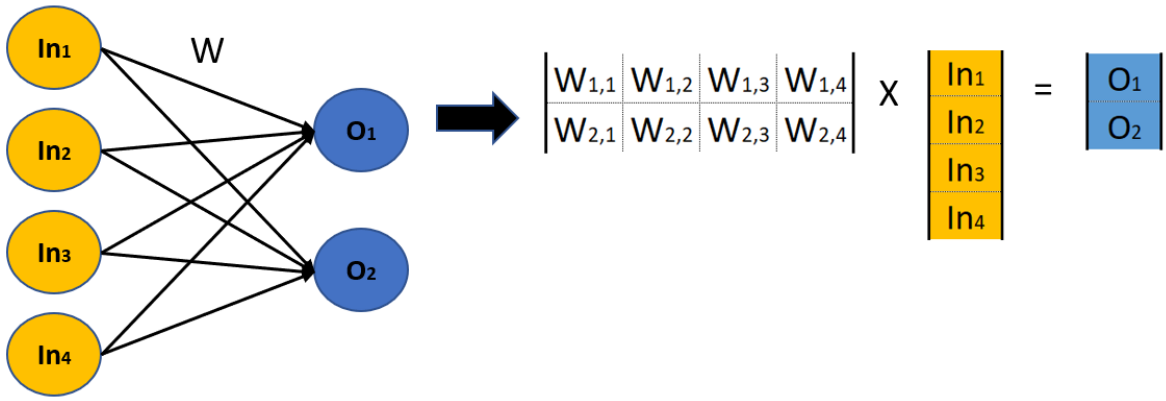
Mapping the fully connected layer into GEMMs is intuitive, as the operation in the layer already follows a matrix-vector multiplication between the matrix of weights and the vector of inputs, as shown in Fig. 3.1a. By employing loop tiling, the problem can partitioned into slices, where we compute on batches of input neurons at a time, as shown in Fig. 3.1b.

Pseudocode 3

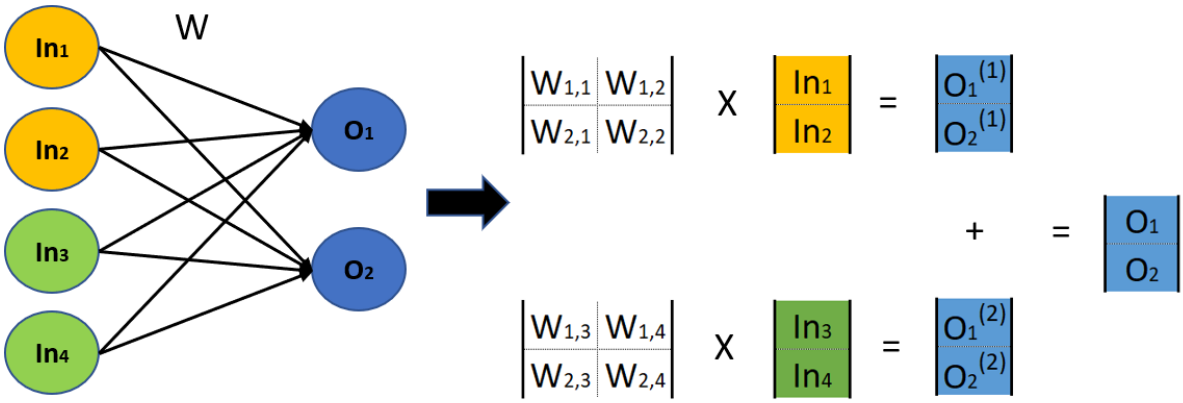
```
original algorithm
for(i=0; i<N; ++i)
   for(j=0; j<M; ++j)
       y[i] += x[j] * w[i][j]

tiled algorithm
for(i=0; i<N; ++i)
   for(k=0; k<M; k+=B)
      for(j=k; j<min(N, k+B); ++j)
          y[i] += x[j] * w[i][j]
```

The work of [5] demonstrates that 3D convolutions of the convolution layer can similarly be mapped to GEMMs. GEMM transformation of the convolution layer involves vectorizing each K×K kernel and then concatenating them together to form weight matrices. Input matrices are generated by vectorizing and rearranging input feature maps. In parallel with convolution, where K×K windowed selections of the input feature map are multiplied



(a) GEMM representation of the fully connected layer (no batching).



(b) GEMM representation of the fully connected layer (with batching).

elementwise by the kernel, each row of the input matrix corresponds to one vectorized K×K windowed selection of the feature map. The calculation of the output feature map follows a GEMM operation between the input matrix and the weight matrix. Fig 3.2 demonstrates an example of this procedure for a single convolution operation. It should be noted the input matrix will contain a lot of repetitive data, since successive rows in the input matrix will have many elements in common. In a hardware implementation, this redundancy may lead to inefficient memory access, where the same piece of input data may be accessed and buffered multiple times. This issue can be remedied through data reuse.

3.1.3 Data Reuse

In CNN acceleration, weights, inputs and intermediate values must be stored in memory until needed. During computation, data is constantly fetched from memory and written back into memory when complete. For FPGAs, memory can be categorized as on-chip or off-chip memory. On-chip memory benefits from very high memory bandwidth but is limited in storage capacity. Off-chip memory has very high storage capacity but has limited memory bandwidth. Data reuse refers to caching data on-chip for fast access and to lower off-chip memory traffic. This is a crucial consideration to balance the ratio of the number of computations versus the number of memory accesses; as relying too much on off-chip memory can potentially cause the overall performance to be bottlenecked by memory access. For example, performing a multiply and accumulate operation involving two inputs and two weights without data reuse will require four read operations and one write operation. In total, the ratio is five off-chip memory access for two multiplies and an add. However, if the next multiply and accumulate have two inputs in common with the first operation, then two memory accesses can be avoided by caching the recurring data. Generally, if no data reuse is employed, memory bandwidth is expected to be a severe hinderance to overall throughput.

3.1.4 Data Quantization

By caching reusable data, we can increase the computation versus memory access ratio. For CNN inference there are many opportunities for reusing inputs and weights. For the convolution layer, the following data reuse strategies can be applied:

- Kernel reuse: during the convolution of an input feature map against a kernel, weights can be read once and reused throughout the entire operation.
- Input window reuse: employing inter-output parallelism will cause the same input feature map to be involved in multiple convolution operations. The same windowed selection of the input can be broadcasted to multiple kernels simultaneously.

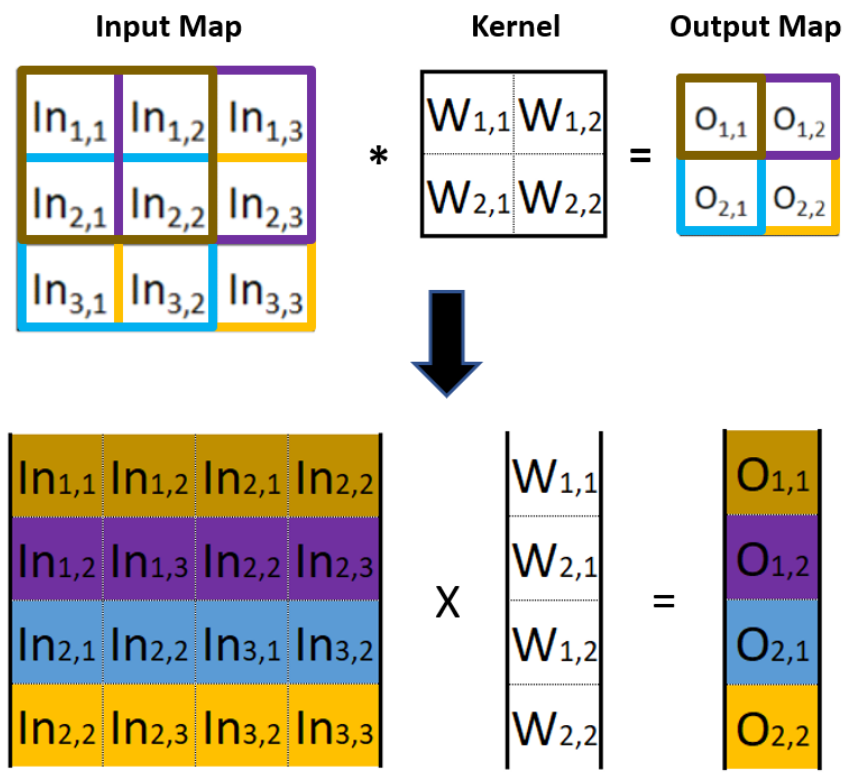


Figure 3.2: GEMM representation of a convolution operation. The color code aims to clarify the mapping between elements in the input and output feature maps to the input and output matrices.

- Caching intermediate results: if loop tiling is used, the entire computation will be partitioned into batches, where the computed results fom each batch represents only a portion of the final result. By caching this intermediate data, we can avoid storing this data off-chip, saving a write and a read access.

While the fully connected layer does not have any opportunities for weight reuse, the second and third strategies can be applied.

The precision of a number representation is affected by the numerical format and the number of utilized bits. The fixed-point format expresses real numbers using binary format, dedicating m bits for the integer and f bits for the fraction. With fixed-point numbers, the

dynamic range and the precision is expanded by increasing the word length. The floating-point format formulates numbers using the mantissa A and the exponent B. With the same word length, the floating-point representation can represent higher dynamic range than a fixed-point counterpart. However, floating-point numbers are constrained to 32-bit and 64-bit representations, while fixed-point numbers can take on various word lengths.

Compared to floating-point, fixed-point arithmetic is more efficient in hardware utilization and power consumption. FPGAs contain high performance DSP blocks that implements 18×19 bit fixed-point multiplication with no logic overhead. To implement 32-bit floating-point multiplication, one DSP block and additional look-up tables are consumed. Overall, fixed-point arithmetic is preferred in applications where fixed-point precision is acceptable.

Past literature has studied the viability of fixed-point representation of weights and inputs for CNN training and inference in comparison to a 32-bit floating-point baseline. [39] demonstrated that CNNs can tolerate reduced precision with negligible change in performance.

3.2 Pruning

Trained neural networks often contain many low-impact parameters that can be discarded without notable degradation in network accuracy. Sparsity regularization [29] is used during training to incentivize certain weights towards being zero valued. This is accomplished during training by using a regularization term in the cost function. Commonly the L1-norm is applied, producing few large values and many near-zero values. Intuitively, zero-valued weights contribute nothing to the output and thus can be effectively ignored. The drawback, however, is that there is little control over where the near-zero weights will reside.

Similarly, connection pruning [24] can be applied on a conventionally pre-trained network, where the importance of every weight is judged based on a selection criterion and

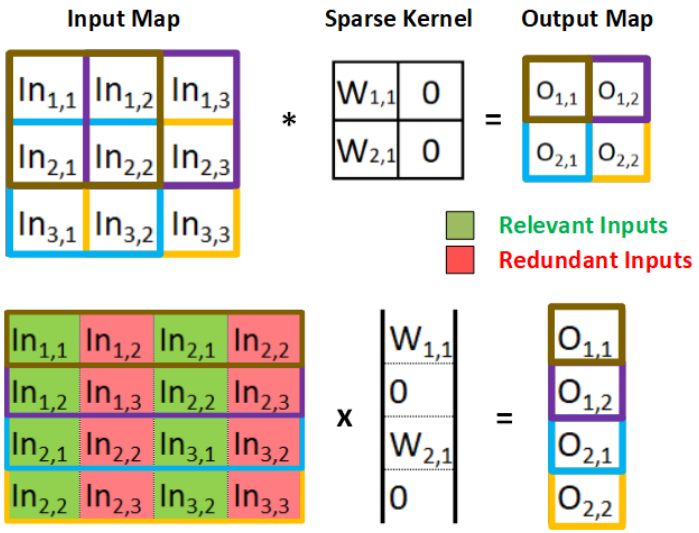


Figure 3.3: GEMM representation of a sparse convolution operation. Relevant inputs contribute to the output, while redundant inputs do not.

ranked. The highest-ranking weights are deemed important and are kept, while the rest are removed. The absolute magnitude of the weight is a simple yet effective criterion.

For both techniques, the resulting network is denoted as sparse since retained connections are irregular and kept weights are scattered throughout the network model. Previous researchers employed fine-grained pruning [22] to remove individual weights, achieving a theoretical 9 times memory and 3 times computational reduction on AlexNet without loss in accuracy. While fine-grained pruning is proven to reduce network size, the irregular weight distribution of the resulting sparse model makes it difficult to attain practical savings in hardware.

Sparse CNN computations can similarly be represented as GEMM operations. As an example, Fig. 3.3 depicts convolution involving a sparse kernel in GEMM format. In this figure, values in the input matrix are highlighted as relevant or redundant. A relevant input is a piece of data that is multiplied by a non-zero weight, thus contributing to the output; while a redundant input denotes one that is multiplied by zero. It is easy to visualize the benefits of pruning, as half of the computations in this example involve redundant inputs

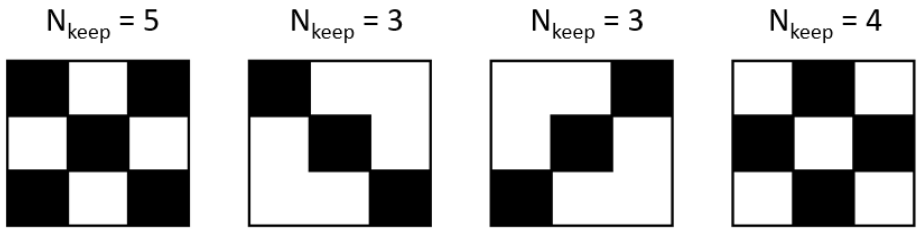


Figure 3.4: Examples of possible intra-kernel stride pruning patterns.

and can be ignored. However, deploying a CNN with irregularly sparse network models in hardware is a challenge, due to variation of the number and location of non-zero elements between kernels. Sparse GEMM computation is a common problem that has been heavily researched. Some solutions have proposed the introduction of regularity in sparsity, where the locations of non-zero values in sparse matrices follow a pattern. This examination into regularization may prove useful in sparse CNN acceleration.

The concept of intra-kernel strided structured sparsity is presented in [6], which is a technique that prunes in accordance to preselected patterns. These patterns follow a rigid structure, for example as shown in Fig. 3.4. The sparse kernel is produced by applying the pruning pattern on the unpruned kernel and resulting a matrix containing N_{keep} values, where N_{keep} is the number of non-zero values in the sparse kernel. Since the data-locality of kernel structures is the same across all kernels, their storage in memory requires only N_{keep} spaces, as shown in Fig. 3.5. While this approach promises efficiency in hardware, the imposed pruning scheme does not remove weights based on importance, but rather location of weights within kernels. These restrictions are harsh and lower the accuracy achievable for a given sparsity.

A compression technique called Compressed Sparse Row (CSR) storage is used in [21] to store the sparse kernels. Unlike intra-kernel strided sparsity, CSR storage can be applied to generic sparse matrices. As shown in Fig. 3.6, CSR storage represents the target sparse matrix in CSR format using two arrays. The weight array stores the individual nonzero elements, the relind array records the relative index of the next non-zero value. Only non-zero weight values and their respective locations are stored, enabling the dense storage of

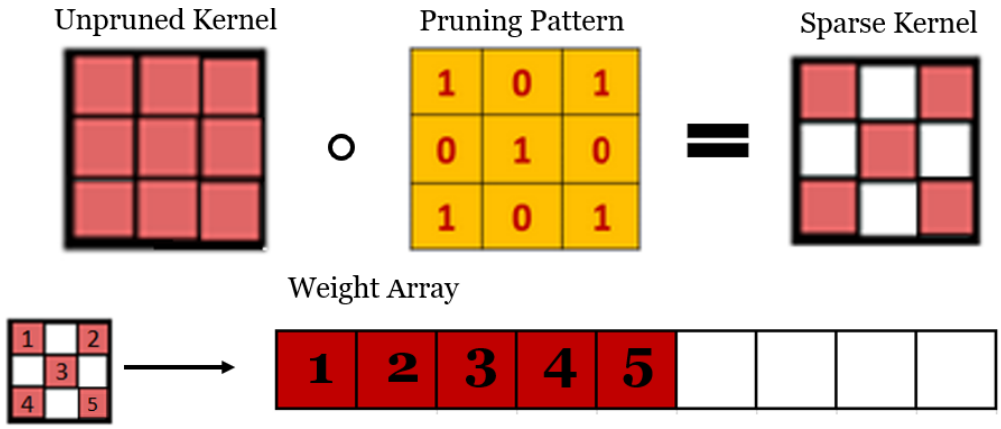


Figure 3.5: Example of the pruning and storage arrangement of intra-kernel strided kernels. Requires N_{keep} units of space.

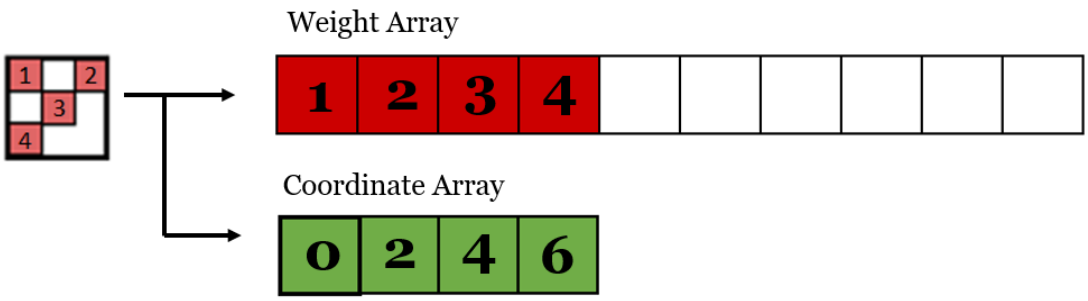


Figure 3.6: Example of the storage arrangement of compressed sparse row. Requires 2×N_{keep} units of space.

highly irregular structures. It is observed that N_{keep} extra space must be allocated for the indices, totalling $2 \times Nkeep$ values.

Coarse-grained pruning [36] was proposed as an alternative pruning method, where entire vectors, kernels [7] or filters [33] are removed. Although pruning at coarser granularities generates more structured models with better data locality, it is more destructive and does not achieve the same performance as fine-grained pruning [36]. Activation pruning is explored in [8], a technique that prunes neurons according to patterns generated from Linear Shift Feedback Registers. While this method is appealing due to the low hardware cost overhead, the network accuracy degradation from pruning is potentially high since all

activations are equally susceptible to removal regardless of their importance.

3.3 Platform Specifications

A DE5A-net FPGA development board [1] was chosen to prototype our accelerator, as shown in Fig. 3.7. Notably, it contains an Altera Arria 10 GX FPGA (10AX115N2F45E1SG). The DRAM used is a DDR3 2133 4GB SO-DIMM from HyperX. The FPGA is capable of generating of soft IP components, such as the NIOS II processor and the external memory interface (EMIF), using on-chip logic. The soft-core nature of NIOS II enables users to configure the processor based on required specifications, and enables the use of custom instructions and the interfacing with custom peripherals. Its versatility makes it a good candidate for a variety of applications, such as digital signal processing. The EMIF is the intermediate interface between external memory and on-chip logic and should be configured to suit the exact type and specification of the desired memory.

On-chip communication between components is facilitated through an Avalon interface. Commonly used are the:

- Avalon Memory-Mapped Interface: a read/write interface between master and slave.
- Avalon Streaming Interface: a unidirectional connection between two components.

Communication to the NIOS II and EMIF necessitates the use of these protocols.

The DRAM runs at approximately 1GHz at double data rate, which enables two memory accesses per cycle. Since typical FPGA designs usually operate below 250MHz frequency, the EMIF can be operated at quarter rate. In this configuration, four DDR3 memory accesses, equating to 8 total memory accesses, can be executed in the span of one FPGA clock cycle. In total, the maximum of transfer rate of 512 bits between memory in every FPGA cycle. This rate determines the memory bandwidth available in our design.

DSP units are hard blocks native to the FPGA dedicated for arithmetic operation. While custom arithmetic hardware can be synthesised using logic, DSPs are preferable

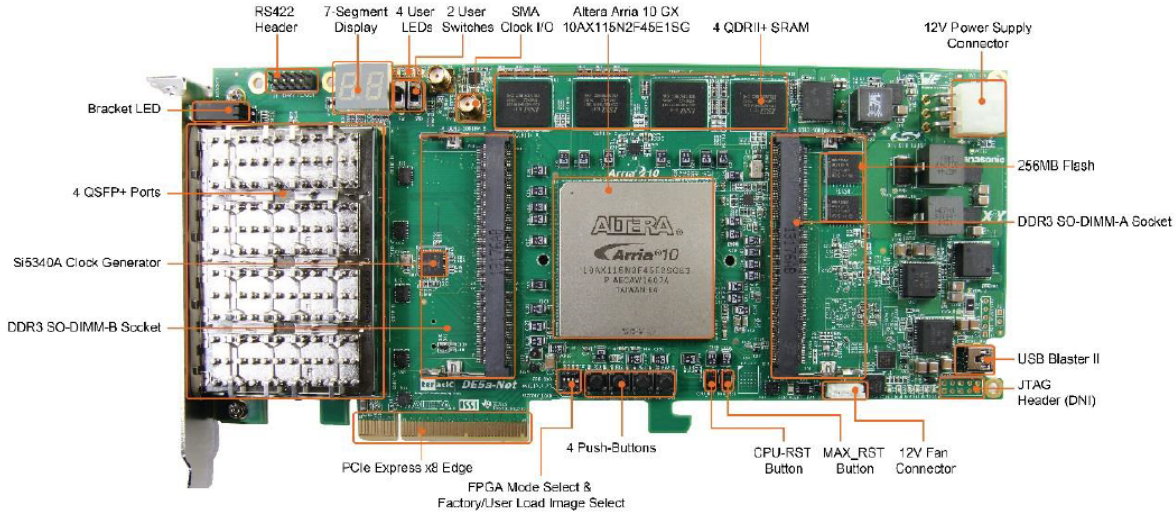


Figure 3.7: Top view of the DE5a-net board [1]

for operations such as multiplication. The Arria10 FPGA family contains DSP units that are optimized for arithmetic and operate at much higher frequency than FPGA logic, and are capable of performing 18×19 precision multiply and accumulations in 1 clock cycle under frequencies below 400MHz. This specification is difficult to achieve using custom synthesized multipliers. However, the number of DSPs is limited; varying from tens of DSPs in low grade FPGAs to thousands in high grade FPGAs. Since active DSPs represent the computational capacity of the hardware, it is crucial minimize their idle time in order to attain high computational efficiency.

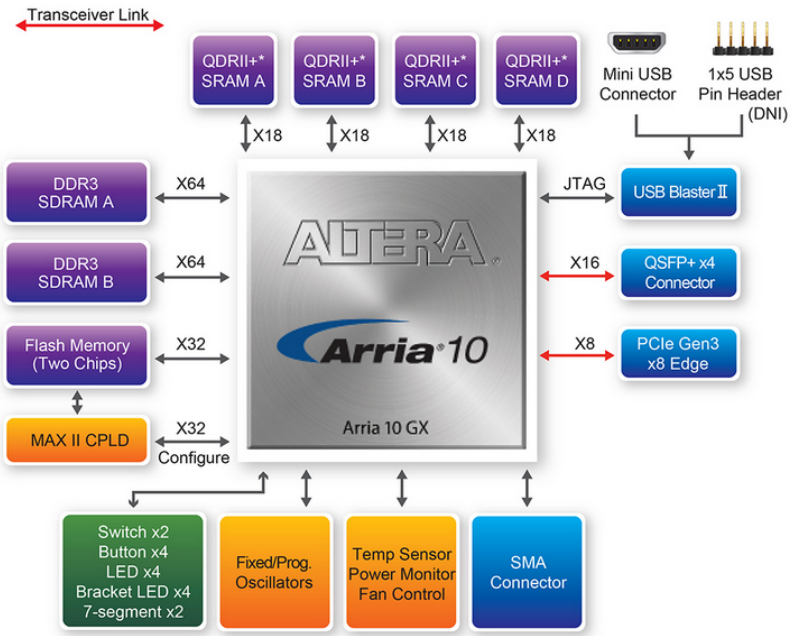


Figure 3.8: Block diagram overview of the DE5a-net board [1]

Chapter 4

Sparse Convolutional Neural Network Acceleration

This section details our proposed intra-kernel regular (IKR) pruning scheme. Section 4.1 describes the procedure of IKR pruning and the structure of IKR sparse CNNs at the algorithmic level, Section 4.2 outlines the system architecture of the hardware accelerator.

We propose a software-hardware co-design principle to accelerate sparse CNNs. On the algorithmic level, we introduce Intra-Kernel Regular (IKR) pruning, a hardware-aware pruning scheme to reduce memory and computational cost for CNN inference. This approach uses generated pruning patterns to preserve important weights while eliminating insignificant weights at the intra-kernel level. Our approach reaps the benefits from fine-grained pruning while maintaining predictable kernel patterns that can be exploited using specialized hardware. Moreover, the resulting sparse kernels can be stored very compactly in compressed sparse pattern (CSP) format, a representation that exclusively keeps non-zero weights and the corresponding mask index. The storage of a kernel requires $N_{keep} + 1$ units of storage, an improvement on the CSR representation. On the hardware front, an FPGA-based accelerator architecture is designed to perform inference on IKR sparse CNNs. This design aims to exploit the model sparsity by avoiding redundant computations involving zero-valued weights.

4.1 Intra-Kernel Regular Pruning

IKR pruning structurally eliminates weights at an intra-kernel level while retaining original accuracy. The proposed scheme supports pruning in the convolutional and Fully Connected (FC) layers. Prior to pruning, a neural network is conventionally trained and acts as the baseline. The trained network model is extracted and kernels with similar locality are grouped into sets. We define a network with m layers, such that the set of layers is $L = \{l_1, l_2, \ldots, l_m\}$. The ℓ-th layer, l_ℓ, has N^ℓ_{sets} sets of kernels such that $l_\ell = \{S^\ell_1, S^\ell_2, \ldots, S^\ell_{N^\ell_{sets}}\}$. Each set of kernels S^ℓ_i, where $i = 1, 2, \ldots, N$, includes N^ℓ_{ker} kernels such that $S^\ell_i = \{W_1, W_2, \ldots, W_{N^\ell_{ker}}\}$. The j-th kernel belonging to S^ℓ_i is denoted as $W^\ell_{i,j}$.

Pruning patterns indicate the locations at which parameters should be kept or eliminated. When pruning at fine-granularity, these patterns are applied at the kernel level, specifying the individual weights that should be removed. The resulting kernel structure is described as irregular since the locations of kept weights are random. Similarly, IKR pruning operates at a fine-grained level; however, we challenge irregularity by memorizing the specific pruning pattern applied to each kernel, allowing us to recall the exact location of kept weights. To reduce storage costs, we impose a restriction on the number of possible pruning patterns. Specifically, for each S^ℓ_i, we have N^ℓ_{pat} possible pruning patterns, $C^\ell_i = \{p_1, p_2, \ldots, p_{N^\ell_{pat}}\}$. A pattern belonging to C^ℓ_i is denoted as $p_{i,k}$, where $k = 1, 2, \ldots, N^\ell_{pat}$.

The objective of pruning is to maximally reduce the number of parameters in the network model while suffering minimal network damage; therefore, it is vital for pruning patterns to retain important weights. We gauge the suitability of a pattern $p_{i,k}$ to a kernel $W_{i,j}$ using the quality metric, $Q(p_{i,k}, W_{i,j})$, and use the highest quality pattern-kernel pair during pruning. This process is explained in more details in Section 4.1.1. The resulting sparse model is retrained to regain the baseline accuracy. Fig. 4.1 illustrates the mechanism for the IKR pruning.

Pruning in the FC layer follows the same methodology that is formerly outlined. The preface for IKR pruning involves grouping kernels into sets. Although connections in

the FC layer are instead represented by a matrix of individual weights, kernels can be artificially created. For example, by grouping 16 parameters, a 4×4 kernel is formed. The IKR pruning follows naturally thereafter.

4.1.1 Mask Pattern Generation

Pruning severs connections within the CNN, reducing the number of learnable parameters and damaging its ability to correctly perform classification. A crucial step during pruning involves determining which parameters can be removed with least affect on network performance. Previous research [22] [33] assigns the importance of a weight to its magnitude. Consequently, pruning patterns that retain a high absolute summation are characterized as having high quality. Alternatively, [29] [7] assess pruning patterns by first applying the pattern and then evaluating the drop in misclassification rate (MCR) on the validation set. Patterns resulting in the smallest MCR drop are considered to be least damaging. Since both methodologies produce comparable performance, the magnitude-based approach is adopted in this work.

Each pruning pattern is represented by a mask of the same shape as the kernels it is targeting. Elements within the mask are either zero-valued or one-valued, with zero representing a prune and one representing a keep. A mask is applied to a kernel via element-wise matrix multiplication, producing a masked kernel. The suitability of a pruning pattern $p_{i,k}$ to a kernel $w_{i,j}$ is determined by the quality metric $Q(p_{i,k}, w_{i,j})$, which is expressed in (4.1). The highest quality pattern for the kernel $W_{i,j}$ is found by an exhaustive search through C_i^ℓ, as illustrated in Fig. 4.2. During pruning, the pattern is permanently applied by overwritting the original kernel with the masked kernel. In consideration of the hardware, each pruning pattern in layer l_ℓ keeps the same number of weights N_{keep}^ℓ.

$$Q(p_{j,k}, w_{i,j}) = \sum |p_{j,k} \odot w_{i,j}| \tag{4.1}$$

Pruning pattern collections are populated through a candidate selection process. Ten promising pruning patterns are generated for each kernel in S_i^ℓ, each of which retains a different permutation of top valuable weights. These patterns are potential candidates

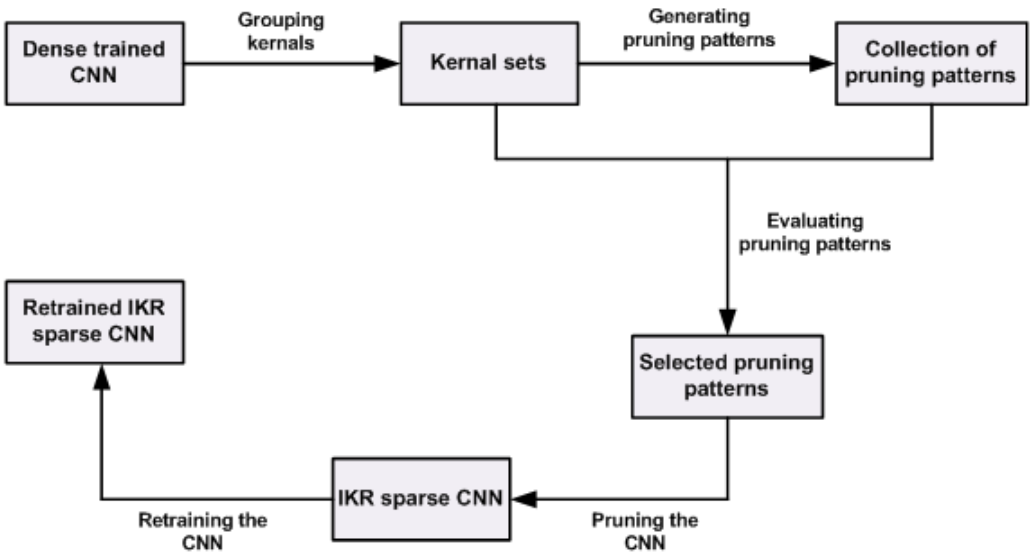


Figure 4.1: Schematic depiction of IKR scheme.

for inclusion into C_i^ℓ. It should be noted that although a pattern may be suitable for a particular kernel $W_{i,j}$, it may not suit other kernels in S_i^ℓ. Since the population of C_i^ℓ is limited to only N_{pat}^ℓ, candidates with the best representation of S_i^ℓ should be chosen. From the entire set of promising pruning patterns generated from S_i^ℓ, N_{pat}^ℓ candidates with the highest overall quality are selected to populate C_i^ℓ.

4.1.2 Layer Sensitivity

Pruning on each layer of the CNN has a different impact on the network performance. Certain layers are tolerant to weight removal and can achieve high sparsity without significant loss in accuracy, while others are more sensitive. Following the approach in [33], we investigate the sensitivity of each layer to pruning. Starting with an original dense model, each layer is isolated and pruned with incrementally higher degree of sparsity, and validation accuracy is recorded at every step. Based on observed sensitivity, we empirically choose how aggressively each layer is pruned by choosing the number N_{keep}^ℓ. For example, sensitive layers are chosen to have a larger N_{keep}^ℓ. Fig. 4.4 shows the network accuracy as each

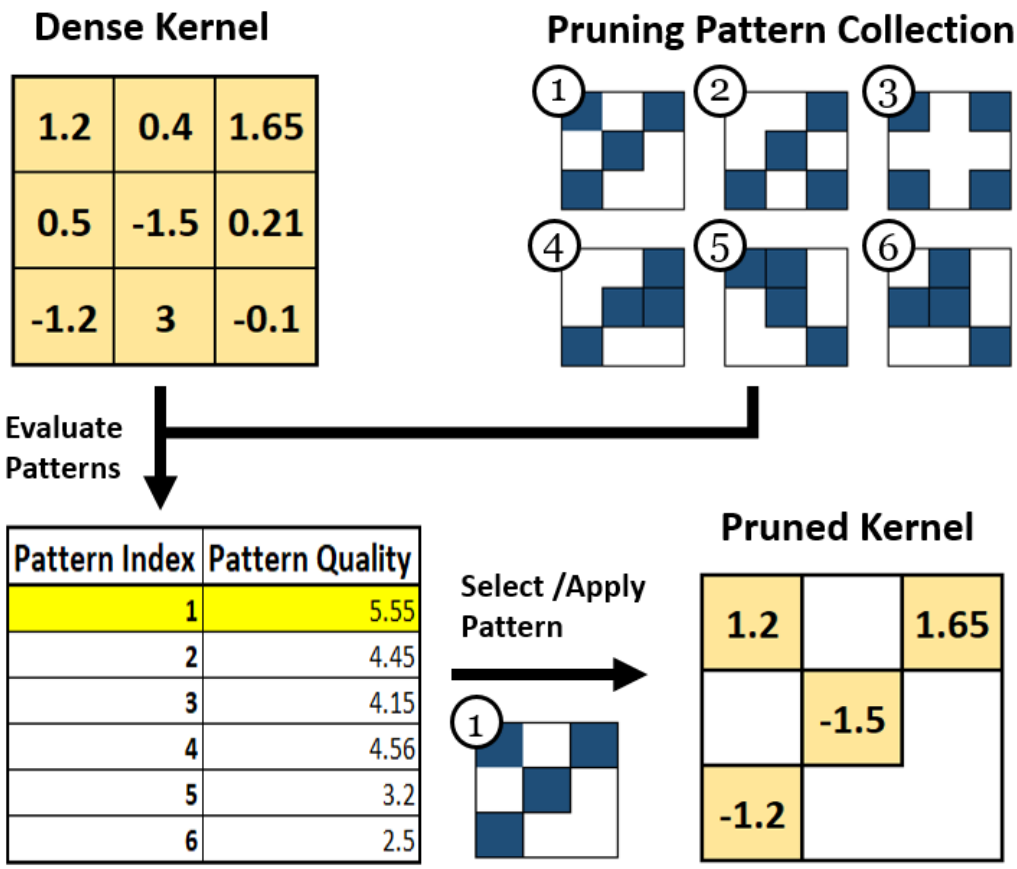


Figure 4.2: Selecting a pruning pattern based on quality (sum of the absolute kept values).

layer is individually pruned for VGG-8 (i.e., the VGG-8 is a VGG-16 [43] inspired CNN containing 6 convoutional and 2 FC layers operating on the CIFAR-10 dataset, adopted from [7]). It is observed that accuracy suffers the most when pruning the first two stages. To explore the impact of N_{pat} on network accuracy, simulation is conducted using the second convolutional layer of VGG-8 as a reference. With the other layers untouched, MCR is measured for various values of N_{pat}^2 at various sparsity. It can be observed from Fig. 4.3 that increasing N_{pat}^2 beyond the value of 8 gives diminishing returns.

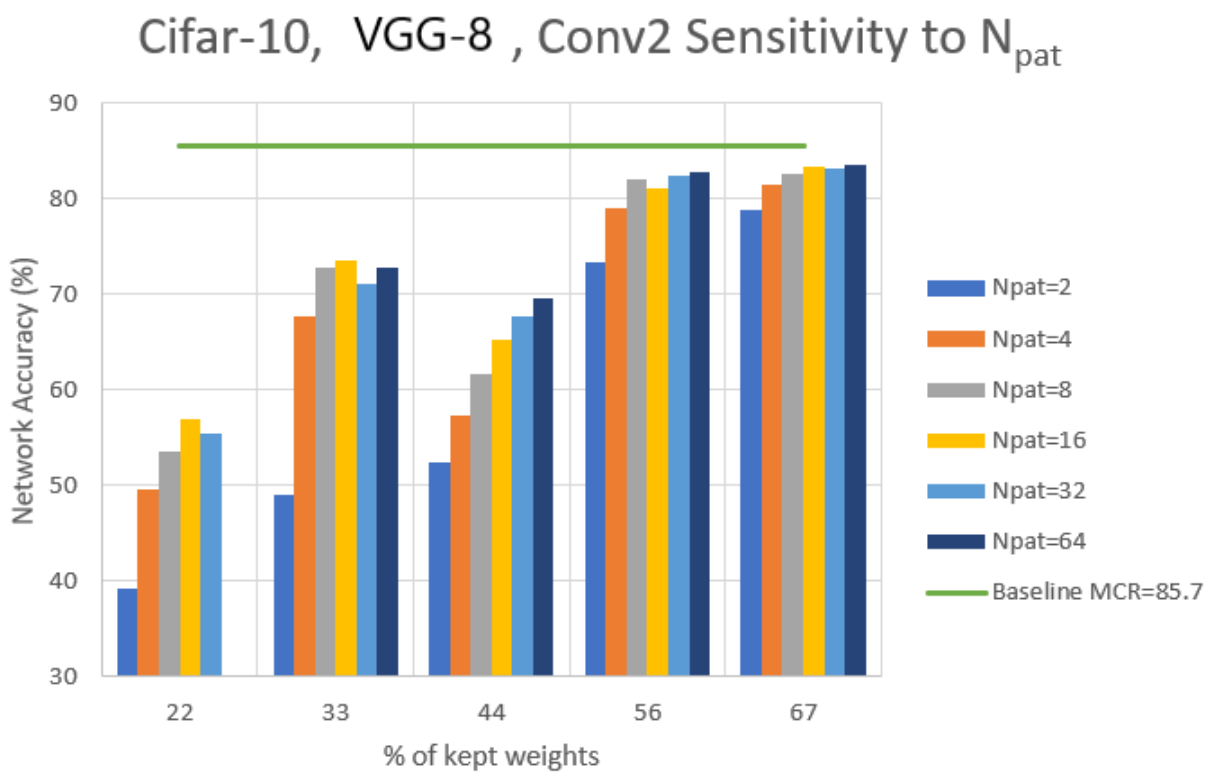


Figure 4.3: The effect of pruning the second convolutional layer using various values of N^2_{pat} on the accuracy of VGG-8 at various sparsities.

4.1.3 Storing Sparse Matrices

To obtain practical savings, the resulting sparse matrices must be stored in a dense format. The work of [21] stores the sparse matrices using Compressed Sparse Row (CSR) notation, a representation that only keeps non-zero weights and their respective indices. We propose a similar format called Compressed Sparse Pattern (CSP) to store IKR sprase kernels. Leveraging that 1) kernels within the same layer keep the same number of weights after pruning, 2) pruning patterns determine the locations of kept weights within each kernel and 3) only N^{ℓ}_{pat} pruning patterns are accessible for each kernel within S^{ℓ}_{i}, CSP exclusively keeps non-zero weights and the corresponding mask pattern index. The stor-

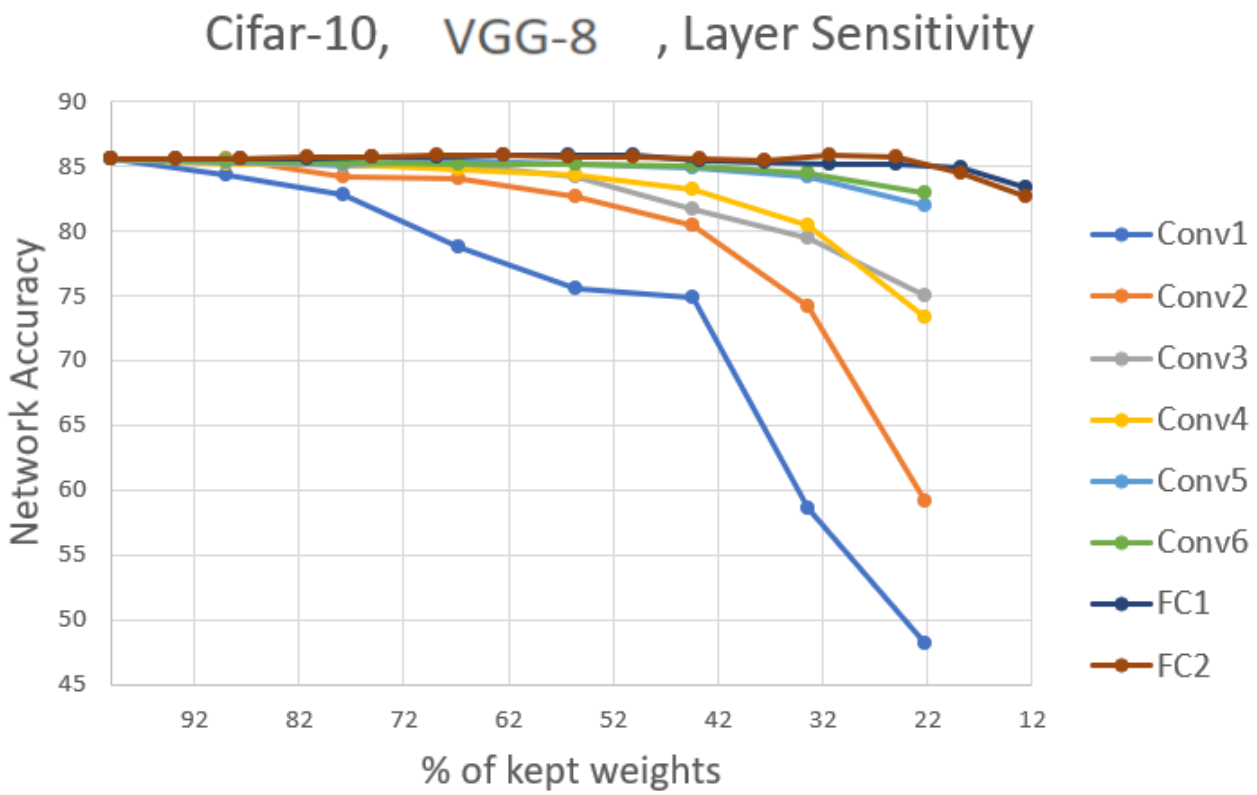
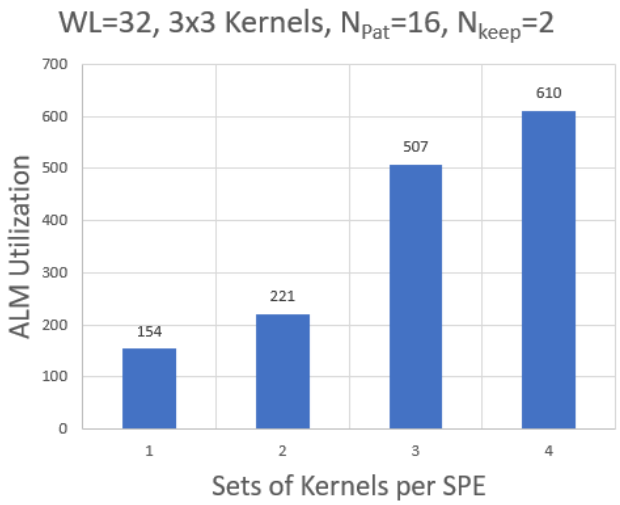


Figure 4.4: Sensitivity to pruning of Convolutional and FC layers from VGG-8. Conv1 represents the first convolutional layer, FC1 represents the first fully connected layer.

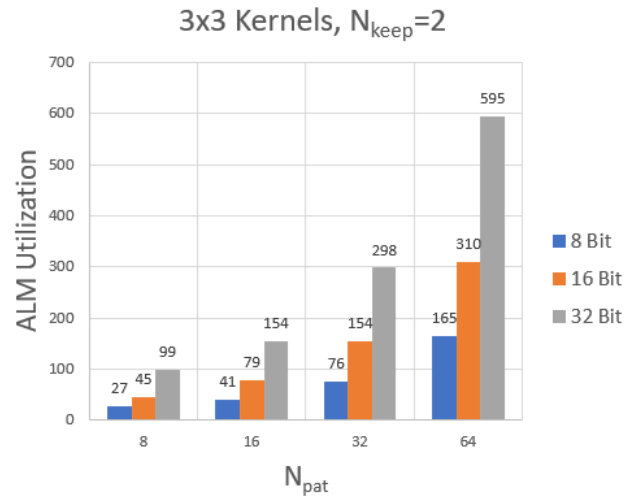
age of a kernel requires $N_{keep} + 1$ units of storage, an improvement on the $2 * N_{keep}$ of CSR representation.The number of bits required to represent the pattern index is equal to $\log_2 N^{\ell}_{pat}$.

4.1.4 Insight on Sparse Computation in Hardware

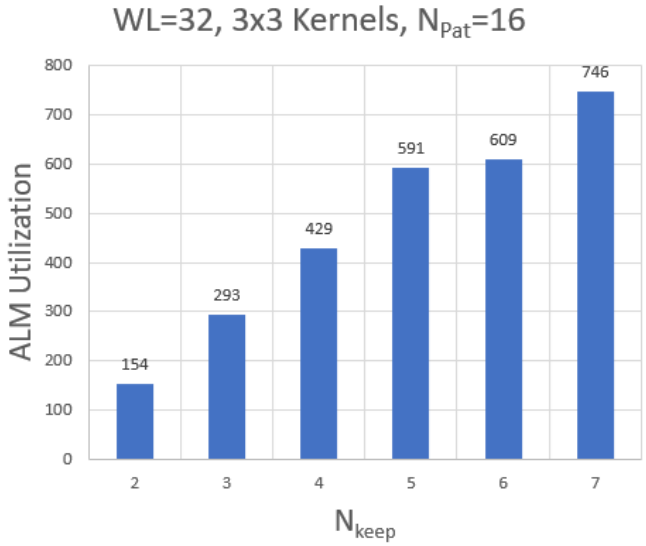
It is difficult to exploit irregular intra-kernel sparsity in hardware since the locations and the number of non-zero weights vary between kernels. As previously mentioned, [21] challenged irregular sparsity by storing non-zero weights in CSR format, transforming irregular structures into regular representations. We propose an alternative approach, where we prune



(a) Varying set coverage.



(b) Varying N_{pat} for 8-bit, 16-bit and 32-bit word lengths.



(c) Varying N_{keep}.

Figure 4.5: ALM utilization for the pattern selector module, with respect to an increasing demand for (a) set coverage, (b) pattern coverage and (c) number of kept weights.

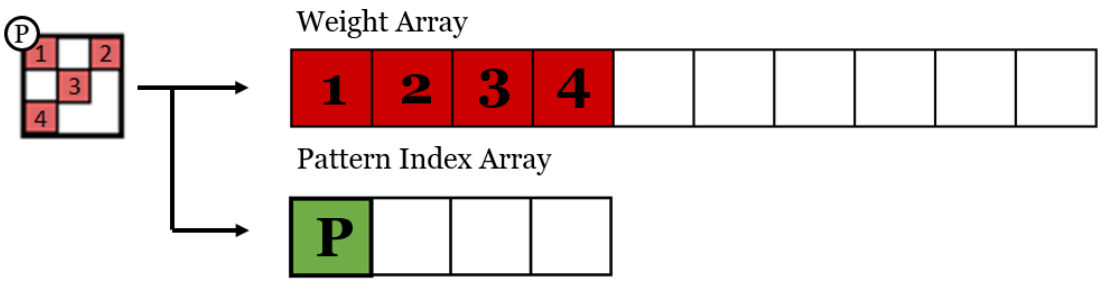


Figure 4.6: Storage arrangement for CSP storage. Requires $N_{keep} + 1$ units of space.

with regularity in mind. IKR pruning restricts the variability in the composition of kernels because the number of pruning patterns is limited. Futhermore, by storing kernels in CSP format, exact composition of every kernel is known. It is expected that an IKR sparse accelerator can employ optimizations targetting traditional CNN structures found in the literature.

4.2 Hardware Implementation

4.2.1 System Overview

We propose an FPGA-based design to accelerate IKR sparse CNNs. Fig. 4.7 shows a general overview of the architecture. The external memory interface (EMIF), the controller and the compute engine are implemented on the FPGA logic fabric. Input/output buffers use on-chip memory to prepare data for processing and to store results. The sparse network model is also stored on-chip, in memory denoted by CSP storage. Input images and output feature maps are offloaded to external DRAM through the EMIF. The compute complex handles most of the computations in the network, including operations for the convolutional, FC, pooling and activation layers

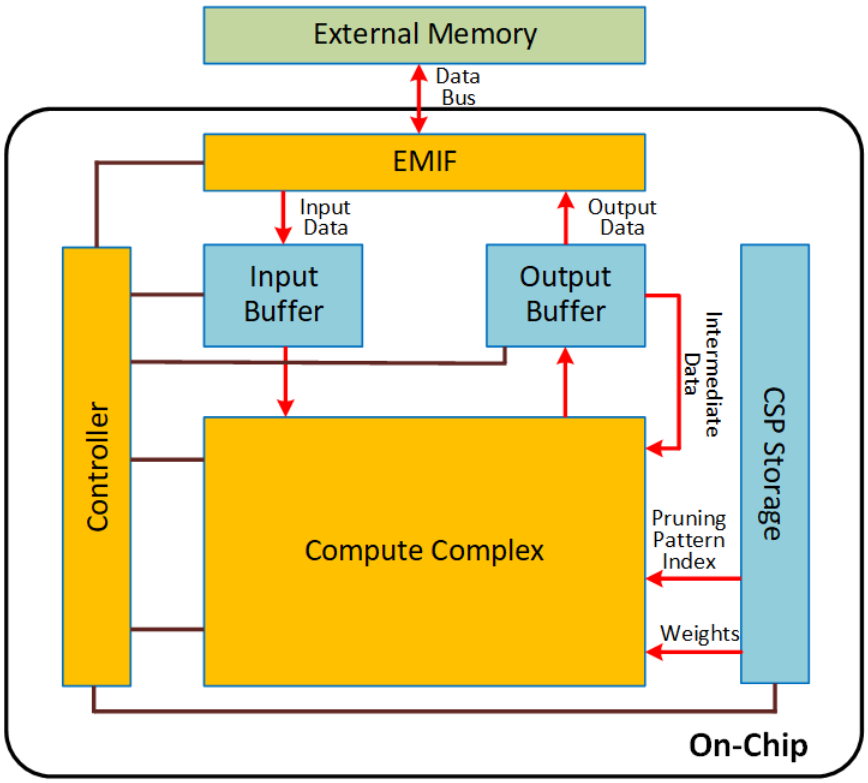


Figure 4.7: Overview of the hardware architecture. External memory denotes the DRAM. The external memory interface (EMIF) facilitates data transfer between the DRAM and the FPGA. Input and output buffers provide temporary storage of values. The CSP storage is on-chip memory that contains weights and pruning patterns. The compute complex contains hardware used for computation. The controller generates signals that directs each block.

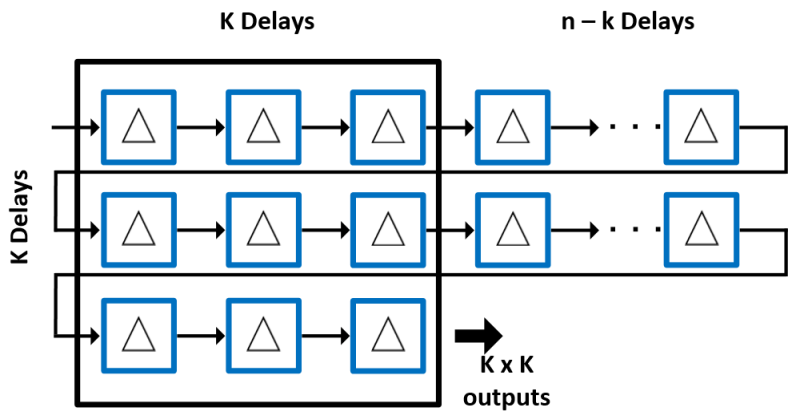


Figure 4.8: Design of the classical line buffer, where the input is streamed in one pixel at a time and the output is a window of $K \times K$ pixels.

4.2.2 Compute Complex

The compute complex contains multiple processing engines (PEs) working in parallel, producing outputs that are further processed by the adder tree and the pooling and activation module. As shown in Fig. 4.9, the compute complex receives data from the input buffer and weights from CPS storage, and passes computed data to the output buffer.

The fundamental role of the PE is to carry out the GEMM operations mentioned in Section. 3.1.2. It comprises a line buffer and multiple sparse processing elements (SPEs). The line buffer, inspired from the convolver design in [9], prepares data for computation. Shown in 4.8, it is comprised of a long row of delay elements that shifts data. During operation, the input feature map is streamed in row-major order at one pixel per clock cycle. As new pixels enter, the old pixels are propagated down the line buffer delay chain. After a preliminary delay of $(K-1)*N+K$ cycles, the line buffer begins to produce valid outputs. The first valid output represents the first windowed selection of the input feature map, which corresponds to the first line in the input matrix. Each subsequent cycle produces the next valid output, representing the next windowed selection of the input feature map, or the next line in the input matrix.

SPEs are responsible for extracting relevant inputs and carrying out the elementwise

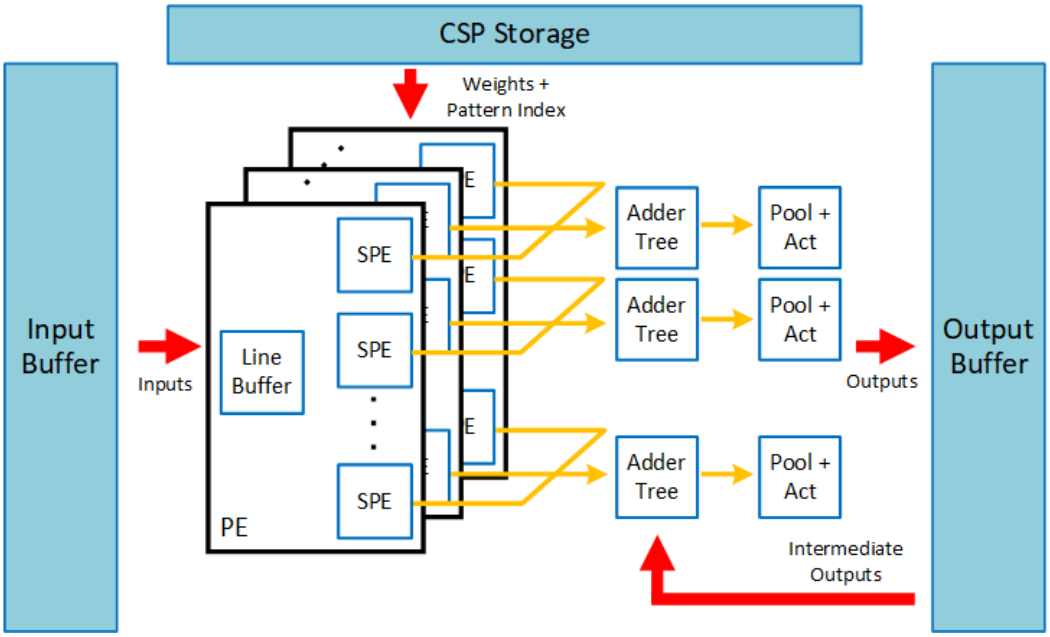


Figure 4.9: Overview of the compute complex. Before operation, weights and patterns are read into the complex. The input buffer streams pixels from input feature maps to the PEs. The output buffer stores the intermediate and final results.

MAC operations in GEMM operations. In preparation for convolution, the SPEs loads the relevant weights and the pattern index corresponding to the kernel from CSP storage and waits for the line buffer to release valid data. When a valid window is received by the SPE, it undergoes a selection process by the pattern extractor, where the relevant inputs are retained while irrelevant inputs are discarded. The relevancy of inputs is determined using the pattern index. This index signifies the kernel's pruning pattern, which is used to infer the location of relevant inputs. The relevant inputs are then multiplied with corresponding weights and accumulated to produce the output. To implement inter-output parallelism, multiple SPEs can be generated within each PE, where the single line buffer broadcasts the windowed selection to all SPEs. Moreover, intra-output parallelism is realized by generating multiple PEs, each of which operates on a seperate input feature map.

Each adder tree is connected to an array of SPEs across all PEs, as shown in Fig. 4.9. Its role is to tie the contributions of each PE together, by summing the partially calculated

results into one value. If the number of input feature maps exceed the number of PEs, then the summed result represents a slice of the final result. In this case, the partial result is buffered as intermediate data until it is summed with the next slice. Once all the slices are accumulated, the output is complete and ready for DRAM storage. The pooling and activation modules are enabled only for the last round of computation. The activation module applies the ReLU function to the incoming data. The pooling module performs max pooling on output feature maps.

Pseudocode 4

```
for(n=0; n<N; ++n):
   for(m=0; m<M; ++m):
      for(r=0; r<R; ++r):
         for(c=0; c<C; ++c):

             for(k=0; k<N_keep_base*N_phase; k+=N_keep_base)
//unroll
                for(j=k; j<min(N_keep_base*N_phase, k+N_keep_base); ++j)
                   output_fm[m][r][c] +=
                      sparse_kernel[n][m][j] *
                      relevant_inputs[n][r][c][j]
```

In traditional CNN accelerators, elementwise MAC operations, for example between kernels and windowed inputs in the convolution layer, are fully parallelised [17] [48]. On the algorithmic level, this involves unrolling the two innermost loops in the convolutional pseudocode seen in Pseudocode 2. In hardware, translates to K^2 multipliers and adders, utilizing K^2 DSPs in implementation. Although this strategy is still valid for sparse CNNs acceleration, it will result in low DSP efficiency in practice. Considering that kernels are sparse, the K^2 DSPs will be assigned many redundant calculations involving zeros. In our design, we aim to parallelize the elementwise MAC operations while maximizing the efficiency of DSPs. IKR pruning imposes restrictions on kernel structures, such that N_{keep} is constant within the same layer and that N_{keep} is a multiple of N_{keep_base}. We

propose partially parallelizing the elementwise MAC operation, such that N_{keep_base} multipliers and adders are synthesized. To calculate the entire elementwise MAC, N_{phase} phases are required, where $N_{phase} = N_{keep}/N_{keep\_base}$. For example, if $N_{keep} = 3 * N_{keep\_base}$ the elementwise MAC operation is completed in three phases. The output from each phase is accumulated to achieve the final result. This strategy is summarized in Pseudocode 4. The DSP efficiency should be high as DSPs will only compute on relevant inputs and non-zero weights.

4.2.3 Data Arrangement and Workload Scheduling

As mentioned in section 3.3, a maximum of 512 bits of data can be transferred between external DRAM and the accelerator. Since we employed 16-bit fixed point quantization, this equates to 32 units of data per cycle. To ensure that the compute engine is efficient, optimizations on the data arrangement of external memory and workload scheduling are employed.

Buffers are necessary to mask the DRAM access latency and to prevent the loss of data. Unlike on-chip BRAM, the latency between issuing read or write instructions to DRAM and the retrieval of valid data is unpredictable. For example, depending on the number of queued instructions, we may experience a variance in latency of 10s of cycles. Shown in Fig. 4.10, FIFOs are used in the input and output buffers to mediate data I/O. Each input FIFO stores incoming data from DRAM and dedicates data to one PE. Each output FIFO stores computed results from one adder tree and pushes the data to DRAM. We define the number of input FIFOs by n_{input_port} and the number of output FIFOs by n_{output_port}. In the design, $n_{input\_port} = n_{pe}$ and $n_{output\_port} = n_{spe}$. Since read and write operations occur in separate phases, the memory bandwidth of the system is $max(n\_input\_port * word\_length, n\_output\_port * word\_length)$.

To decrease DRAM access latency and to minimize PE idle time, an interleaving data storage arrangement is imposed. This strategy operates under two axioms:

- The DRAM should be accessed on consecutive addresses in memory space. According

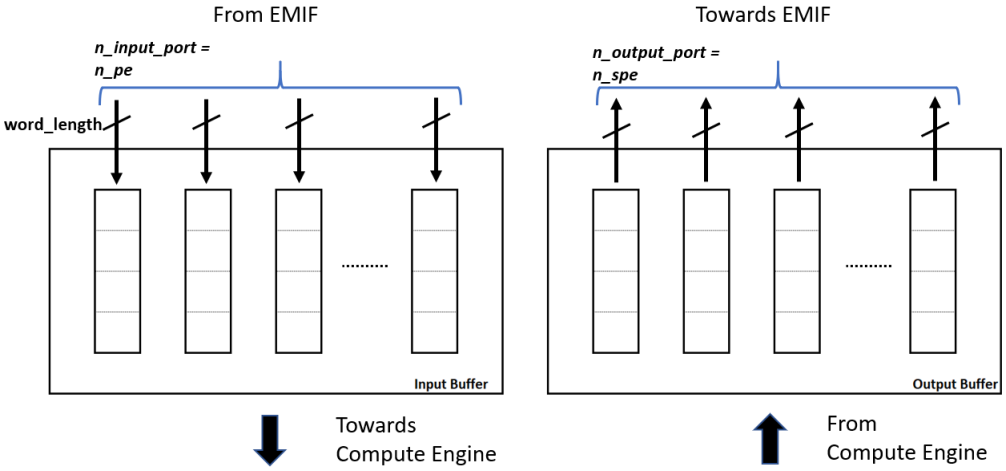


Figure 4.10: The structure of the input and output buffers, consisting of FIFOs. They control the dataflow to and from the accelerator.

to [2], non-consecutive access will require extra set up time, resulting in a longer read latency.

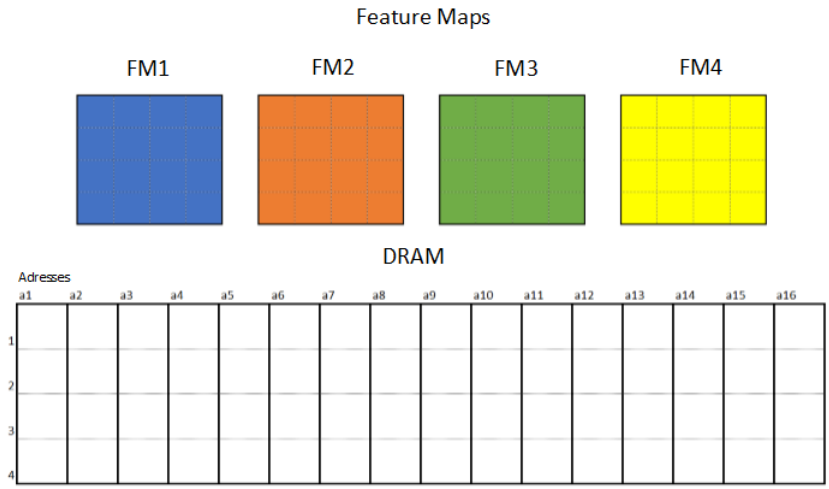
- The rate at which PEs receive data should be balanced. The workload should be divided evenly among PEs, rather than overloaded onto one PE.

To illustrate the thought behind this strategy, we pose the hypothetical problem illustrated in 4.11a, where the goal is to store the four color coded feature maps in DRAM while maintaining both axioms. Fig. 4.11b outlines the naïve strategy of a linear data arrangement, where each input feature map is stored in individual blocks of memory in row-major order. Evidently, this orientation cannot satisfy axiom 1 and 2 simultaneously. If DRAM is read consecutively, each input feature map must be fully read before other input feature map can be accessed. Consequently, one PE will be overloaded with pixels from one input feature map while PEs remain idle. Moreover, a long FIFO will be required to accommodate the queued data. On the other hand, if we attempt to balance the workload of the PEs by reading different input feature maps,each of which reside on non-consecutive addresses, read latency will be high. The data interleaving strategy, shown in Fig. 4.11c, satisfies the requirements. In this approach, each address in memory contains one pixel of

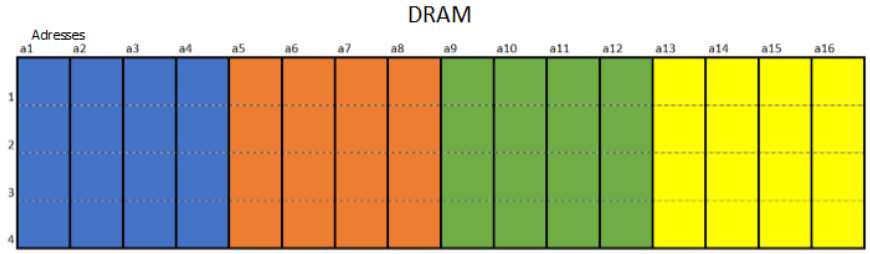
data from each feature map. As a result, each consecutive DRAM read operation accesses a new pixel from each input feature map, which is then stored in the FIFO corresponding to the correct PE. Since data is received simultaneously and steadily, the efficiency of PEs is high.

Since the maximum memory bandwidth is 32 pixels, the number of input feature maps we can interleave is limited to 32. If the number of input feature maps in a layer exceeds this, then only a slice of the total input feature maps can be computed simultaneously, resulting in partially calculated outputs. In order to finalize the output, the remaining slices of input feature maps must be computed. To accommodate this, we impose loop tiling by dividing the workload into batches that are computed in phases, where the intermediate results of each phase are accumulated. Given that N input feature maps are present in a layer and that n_{pe} number of PEs are used for calculation, n_{prime} phases are necessary, where $n_{prime} = ceil(N/n_{pe})$.

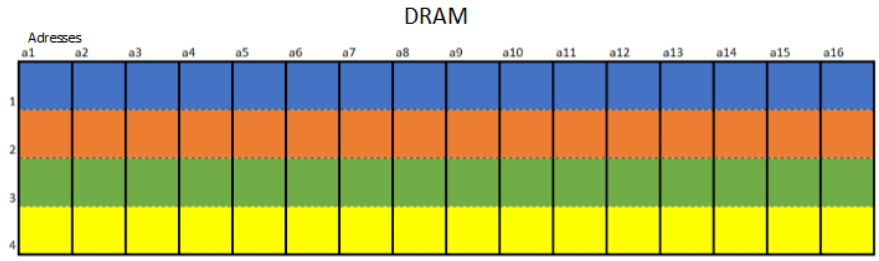
To reduce external memory traffic, reusable pixels are be broadcasted to multiple SPEs for simultaneous calculation. For the convolution layer, each pixel in the input feature map is used in total $M * K^2$ separate occasions. Detailed in Section 4.2.2, the line buffer reuses pixels K^2 times during the spam of convolution, thus reducing repeat data accesses of pixels to M times. By sharing pixels among SPEs, repeat accesses of pixels is further reduced by n_{SPE} times down to m_{prime} times, where $m_{prime} = M/n_{SPE}$.



(a) This illustrates the problem formulation for input feature map storage strategy in DRAM. In this example, the DRAM contains 16 address spaces, each of which accomodates 4 units of storage. Each input feature map contains 16 values, and each value requires one unit of storage. The goal is to store the 4 input feature maps in DRAM.



(b) Depiction of DRAM memory composition under the naïve storage strategy. When accessing consecutive addresses, an entire input feature map is read before proceeding to the next. This subjects some DSPs to an overload of work while others remain idle.



(c) Depiction of DRAM memory composition under the interleaving storage strategy. When accessing consecutive addresses, pixels from each of the 4 input feature maps are simultaneously read. This allows for a balanced work distribution among DSPs.

Chapter 5

Experimental Results

This section presents the results of our experimentation. Section 5.1 outlines the compression rate and accuracy of IKR pruning in software, Section 5.2 outlines the performance of our proposed hardware accelerator.

A software-hardware co-design philosophy is used to design a hardware prototype aimed to accelerate IKR pruned CNNs. In software, we create and train the baseline CNNs models such that their architecture and prediction accuracy are comparable to similar CNNs found in literature. Once trained, the IKR pruning algorithm is applied to the baseline models to create IKR sparse versions of the CNNs. To validate the concept of IKR pruning, the IKR sparse CNN is compared against other pruning techniques found in literature. Once IKR pruning is finalized in software, the CNN model is extracted and saved on the hardware platform in memory. The performance of the hardware is measured and compared against other FPGA accelerators in literature.

5.1 Software

To investigate the validity and performance of the IKR pruning algorithm, simulations were conducted on LeNet-5 and VGG-8 in Python using Tensorflow. The architectures of the two networks are outlined in Table 2.2.

Table 5.1: Parameters used during IKR pruning on LeNet-5.

Layer	Kernel Size	N_{sets}	N_{pat}	N_{keep}
C1(1×20)	5×5	2	8	6
C2 (20×50)	5×5	10	8	2
FC(800×500)	5×5	10	8	2
FC(500×10)	5×5	5	8	2

Table 5.2: Parameters used during IKR pruning on VGG-8.

Layer	Kernel Size	N_{sets}	N_{pat}	N_{keep}
C1(3×128)	3×3	3	8	6
C2 (128×128)	3×3	8	8	4
C3(128×128)	3×3	8	8	2
C4(128×128)	3×3	8	8	2
C5(128×256)	3×3	16	8	2
C6(256×256)	3×3	16	8	2
FC(256×256)	3×3	8	8	4
FC(256×10)	3×3	5	8	4

The networks were trained using Stochastic Gradient Descent (SGD) and Adam optimization with mini-batches of 128 images using 32-bit floating point numbers. For each layer l_ℓ, the parameters N^ℓ_{keep}, $N^\ell_{set}$and N^ℓ_{pat} are empirically chosen in order to maximize network sparsity while incurring minimum loss in network accuracy. The parameters used during IKR pruning of LeNet-5 and VGG-8 are reported in Table 5.1 and Table 5.2 respectively.

5.1.1 LeNet-5 on MNIST

The similation tests were carried out on LeNet-5, comparing the IKR pruning scheme to Fine-Grained Pruning [22] in terms weight and computational reduction. The evaluation

Table 5.3: Pruning statistics for LeNet-5 and VGG-8. FG: Fine-Grained, FMK: Feature Map followed by Kernel.

Pruned Network	Baseline Error	Final Error	Weight Density	Computational Density
LeNet-5 IKR	0.6%	1.1%	10%	13.8%
LeNet-5 FG [22]	0.8%	0.77%	8%	16%
VGG-8 IKR	14.9%	15.8%	23.2%	19.2%
VGG-8 FMK [7]	16.26%	17.26%	25%	-

of the two techniques was performed on the MNIST dataset, where we divided the original training set of 60,000 samples into a 55,000 sample training set and a 5,000 sample validation set. To artificially inflate the training set, the training set is cloned; random contrast and random flip transformations are applied to the replicated images. The CNN was trained for 15 epochs and then for 10 epochs using learning rates of 0.001 and 0.0001 respectively. When retraining the sparse CNN, the learning rate was set at 0.005 for 10 epochs and 0.0001 for 10 epochs.

Table 5.3 outlines the performance of the IKR pruned CNN, showing that the original network size is compressed by 10 times and the required computations is reduced by 7 times. IKR pruning achieves similar compression rates in comparison to Fine-Grained pruning in the literature, with lower the weight reduction but higher computational reduction. The final network error rate is 0.5% higher than the baseline error-rate.

5.1.2 VGG-8 on CIFAR-10

VGG-8 is used to perform classifcation on the CIFAR-10 dataset. The simulation tests compare the IKR pruning to the Feature Map followed by Kernel-Level (FMK) pruning [7] in terms of weight reduction and accuracy. We divided the original training set of 50,000 samples into a 45,000 sample training set and a 5,000 sample validation set. Prior to training, the input images are preprocessed with a whitening transformation. To artificially double the number of training images, the training set is duplicated; and random contrast and random flip transformations are applied to the replicated images. The CNN was trained for 50 epochs and then for 20 epochs using learning rates of 0.001 and 0.0001

Table 5.4: Resource Utilization and parallelism parameters of our accelerator designs.

CNN	Freq(MHz)	ALMs	DSPs	n_{pe}	n_{spe}	N_{keep_base}
LeNet-5 IKR (Ours)	150	33,119	103	5	10	2
VGG-8 IKR (Ours)	150	301,408	1027	16	32	2

Table 5.5: Resource Utilization and parallelism parameters of our accelerator designs.

CNN	Images/S	GOP/Sec	Eff GOP/Sec	Classification Error %
LeNet-5 IKR (Ours)	4185	2.65	19.19	1.1
VGG-8 IKR (Ours)	426	77.7	403	15.8

respectively. When retraining the sparse CNN, the learning rate was set at 0.001 for 20 epochs and 0.0001 for 20 epochs. Table 5.3 shows that the IKR pruning compresses the original network size by 4 times and reduces the required computations by 6 times. It can be seen from the table that with the same 1% accuracy degradation, the IKR pruning achieves slightly higher weight reduction compared to the FMK pruning. While the FMK pruning did not provide computational savings, it is reported for the IKR pruning.

5.2 Hardware

In this section, the performance of our system is evaluated and compared against other FPGA-based CNN accelerators. The hardware prototype was platformed on a DE5a-Net development board, containing the Intel Arria 10 GX (10AX115N2F45E1SG) FPGA, under 16-bit quantization.

Table. 5.5 shows the number of PEs, SPEs and the N_{keep_base} chosen for each accelerator design. These parameters indicate the degree of parallelism in the design and are correlated with the resource consumption on the FPGA. The DSP utilization heavily linked to these parameters as they implement multipliers. It is observed that the number of DSPs used is roughly the product of n_{pe}, n_{spe} and N_{keep_base}, which is also equal to the number of multipliers used in the design. ALMs are generic logic cells that contain resources such as

Table 5.6: Layer by layer performance of accelerator on IKR Pruned LeNet-5

Layer	Required MOP	Execution time (μS)	GOP/Sec	Eff GOP/Sec
C1(1×20)	0.184	35.2	5.22	16.3
C2 (20×50)	0.384	32	12	100
FC(800×500)	0.064	169	0.38	4.72
FC(500×10)	0.0008	2.54	0.313	3.9
Overall		239	2.65	19.19

lookup tables and flip flops. It appears that increasing n_{pe} and n_{spe} each by three times from LeNet's to VGG-8's design results in a proportional increase of ALM usage by nine times.

The performance of the accelerator targeting IKR sparse LeNet-5 is shown in Table 5.6. It processes 4185 images per second at an overall computation rate of 2.65 GOP/S. However, considering that the dense LeNet-5 has a computational complexity of 4.59 MOPs per image while the IKR sparse LeNet-5 has 0.634 MOP, each operation towards the sparse CNN is effectively more useful. Thus, while our system performs an overall 2.65 GOP of work per second, it effectively performs 7$\times$ more, at 19.19 GOP/S. It should be noted that the accelerator is more efficient at calculating convolutional layers than fully connected layers, evident by the huge discrepancy in computational rate. This phenomenon is a result of the convolutional layer having a high work to memory traffic ratio, where each piece of data read from DRAM is reused for many computations (largely from the window buffer). The fully connected layer has a lower ratio and performance in this layer is bound by memory bandwidth. For LeNet-5, it can be seen that the execution time of the fully connected layers far surpasses the convolutional layer. Decreasing the size of the first fully connected layer should substantially boost overall performance.

The performance of the accelerator targeting IKR sparse VGG-8 is shown in Table 5.7. It processes 426 images per second at an overall computation rate of 77.7 GOP/S. Since the computational requirements for the IKR sparse VGG-8 is 19% of the original network, each operation is 5 times more useful. The effective performance is 403 GOP/S. Compared

Table 5.7: Layer by layer performance of accelerator on IKR Pruned VGG-8.

Layer	Required MOPS	Execution time (μS)	GOP/Sec	Eff GOP/Sec
C1(3×128)	6.22	277	22.4	67
C2 (128×128)	154.14	1549	99.4	447
C3(128×128)	14.15	179	79	711
C4(128×128)	9.83	156	63	567
C5(128×256)	3.53	66	53.5	481
C6(256×256)	0.39	89	4.4	40
FC(256×256)	0.0164	23	0.7	5.5
FC(256×10)	0.00064	2	0.3	2.4
Overall		2349	77.7	403

to LeNet-5, the fully connected layers of VGG-8 are much smaller and their execution time accounts for a small portion of the overall. As such, the VGG-8 based accelerator operates largely in the high-performance convolutional layer domain. Thus, with 10 times increase in resource consumption from LeNet-5, the VGG-8 accelerator achieves more than 10 times gain in overall performance. It should also be remembered that the computation reduction, resulting from pruning, of each layer varies depending on N_{keep}. For example, the percentage reduction of C3 is larger than that of C2. This is the reason why layers such as C3 show higher effective performance than layers such as C2 despite having a lower performance.

Table. 5.8 compares the performance and DSP efficiency our designs to FPGA-based CNN accelerators in literature. Since DSPs represent the computational power of the FPGA, its scarcity represents an upper bound in the performance of the accelerator. By achieving a high performance to DSP ratio, we can ensure that these limited resources are used to their highest potential. While our IKR sparse LeNet-5 design eclipses [17] in performance and DSP efficiency, [35] outperforms us in this metric by 4.7 times. While this is true, [35] also uses 6.2 times more DSPs, and our design overall achieves 1.3 times higher DSP efficiency. Our IKR sparse VGG-8 design exceeds the effective performance

Table 5.8: The performance of our accelerators in comparison to other FPGA-based accelerators found in literature.

CNN	Required MOPs	Eff GOP/Sec	DSPs	(Eff GOP/Sec)/DSP	Accuracy
LeNet-5 IKR (Ours)	0.634	19.19	103	0.186	98.9 %
[17]	4.59	0.317	79	0.004	N/A
[35]	4.59	90	638	0.141	N/A
VGG-8 IKR (Ours)	182	403	1027	0.393	84.2 %
[10]	520	16	1056	0.015	N/A
[50]	1330	61.62	2800	0.022	N/A
[39]	30760	137	790	0.173	N/A

of [39] by 3 times. Although [39] uses less DSPs than our design, our DSP efficiency is 7.8 times higher.

Chapter 6

Conclusions

In this work we present structured CNN pruning for efficient hardware implementation and an FPGA-based framework for sparse CNN acceleration. At the algorithmic level, an IKR pruning scheme was proposed to compress CNNs at fine granularity while maintaining structured data-locality. The sparse network model is compressed to reduce memory footprint in preparation for hardware. By applying the IKR pruning to benchmark CNNs, it is demonstrated that IKR pruning scheme achieves high weight and computational sparsity with negligible degradation in inference accuracy. The hardware architecture is designed to accept the compressed network model from software and aims to compute solely on relevant data. Since inference on the sparse model requires less computations than the original network, each computation is effectively more useful. Evaluation on accelerators targeting benchmark CNNs demonstrates successful inference without loss in accuracy.

CNN research is extremely fast paced, with newer architectures making huge strides in performance and utility. Due to the existence of deep learning frameworks such as Tensorflow, exploration and development of CNNs in software are quick and widely accessible. As a result, software has outpaced hardware in many facets of CNN research. However, as design tools mature, it is now possible for FPGAs to be integrated into popular deep learning frameworks. Through these tools, FPGA design can be fast and flexible, expediting the transfer of developing CNN trends into hardware.

A promising avenue of CNN research lies with the exploration of different data types. It has been demonstrated that a reduction of precision from 32-bit floating-point down to 8-bit fixed-point can result in negligible loss in classification accuracy. This reduction not only compacts the storage of neurons and weights but also results in simpler arithmetic hardware. Approximate computing techniques such as Binarized neural networks (BNNs) and ternary neural networks (TNNs) pushes this idea to the extreme, by using 1-bit and 2-bit data types respectively. Interestingly, the computation of arithmetic at this precision maps to simple gate-level operations, which massively reduces the cost of arithmetic hardware. While BNNs and TNNs enjoyed success in the classification of the CIFAR-10 dataset, results have not been realized for larger datasets. Stochastic computation (SC) is a promising low-cost technique that represents data through a generated sequence of bits. Similar to BNNs, SC benefits from extremely cheap arithmetic operations, as they map to simple gate-level operations. For example, multiplication between two SC numbers can be realized by an AND gate. The standout feature of SC is the precision its representation. The precision of SC numbers is determined by the length of the generated sequence, where a longer sequence offers higher resolution. Unlike the binarized representation in BNNs, SC can choose its precision by varying the length of the sequence. However, SC suffers from several weaknesses including a lengthy run-time for computations and the multiplication of near zero values being erroneous. Pruning offers a potential solution to the latter problem since it removes low-magnitude weights. The incorporation of pruning and SC techniques in CNN acceleration may offer a promising avenue of research.

References

[1] De5a-net. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=231&No=970&PartNo=5.

[2] Emif. https://www.intel.com/content/www/us/en/programmable/support/support-resources/external-memory.html.

[3] Intel arria 10 native fixed point dsp ip core user guide. https://www.intel.com/content/dam/altera-www/global/en_US/.../ug_nfp_dsp.pdf.

[4] Hamid Abdel, Mohamed Ossama, Rahman Abdel, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Trans. Audio, Speech, and Language Processing*, 22(10):1533–1545, 2014.

[5] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. Low-memory gemm-based convolution algorithms for deep neural networks. *arXiv preprint arXiv:1709.03395*, 2017.

[6] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):32, 2017.

[7] Sajid Anwar and Wonyong Sung. Compact deep convolutional neural networks with coarse pruning. *arXiv preprint arXiv:1610.09639*, 2016.

[8] Arash Ardakani, Carlo Condo, and Warren J. Gross. Activation pruning of deep convolutional neural networks. *5th IEEE Global Conf. on Signal and Information Processing (GlobalSIP)*, pages 1325–1329, 2017.

[9] Bernard Bosi, Guy Bois, and Yvon Savaria. Reconfigurable pipelined 2-d convolvers for fast digital signal processing. *IEEE Trans. on VLSI Systems*, 7(3):299–308, 1999.

[10] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 38(3):247–257, 2010.

[11] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[12] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Proc. IEEE/ACM Int. Symp. on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[13] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537.

[14] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3123–3131, 2015.

[15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[16] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 92–104. ACM, 2015.

[17] S. Ghaffari and S. Sharifian. Fpga-based convolutional neural network accelerator design using high level synthesize. In *2016 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS)*, pages 1–6, Dec 2016.

[18] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Prentice Hall, Upper Saddle River, N.J., 2008.

[19] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proc. Int. Conf. on Machine Learning*, pages 1737–1746, 2015.

[20] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: efficient inference engine on compressed deep neural network. In *Proc. Int. Symp. on Computer Architecture*, pages 243–254. IEEE Press, 2016.

[21] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding. In *Int. Conf. on Learning Representations 2016*.

[22] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[23] Xushen Han, Dajiang Zhou, Shihao Wang, and Shinji Kimura. CNN-MERP: An FPGA-based memory-efficient reconfigurable processor for forward and backward propagation of convolutional neural networks. In *2016 IEEE 34th Int. Conf. on Computer Design (ICCD)*, pages 320–327. IEEE, 2016.

[24] Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. In *IEEE Int. Conf. on Neural Networks, 1993*, pages 293–299. IEEE, 1993.

[25] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[26] Branislav Kisačanin. Deep learning for autonomous vehicles. In *IEEE Int. Symp. on Multiple-Valued Logic (ISMVL), 2017*, pages 142–142.

[27] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS,2012*, pages 1097–1105, 2012.

[29] Vadim Lebedev and Victor Lempitsky. Fast convnets using group-wise brain damage. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 2554–2564, 2016.

[30] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. of the IEEE*, 86(11):2278–2324, 1998.

[31] Yann LeCun, Lon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324, 1998.

[32] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[33] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[34] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *26th Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–9. IEEE, 2016.

[35] Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. Automatic code generation of convolutional neural networks in fpga implementation. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 61–68, Dec 2016.

[36] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

[37] Marvin Minsky and Seymour Papert. *Perceptrons - an introduction to computational geometry*. MIT Press, 1987.

[38] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress.

[39] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded FPGA platform for convolutional neural network. In *Proc. ACM/SIGDA Int. Symp. on FPGAs*, pages 26–35. ACM, 2016.

[40] Ao Ren, Zhe Li, Caiwen Ding, Qinru Qiu, Yanzhi Wang, Ji Li, Xuehai Qian, and Bo Yuan. SC-DNN: Highly-scalable deep convolutional neural network using stochastic computing. In *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 405–418. ACM, 2017.

[41] Sebastian Ruder. An overview of gradient descent optimization algorithms., 2016. cite arxiv:1609.04747Comment: Added derivations of AdaMax and Nadam.

[42] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–, October 1986.

[43] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[44] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-Sun Seo, and Yu Cao. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proc. ACM/SIGDA Int. Symp. on FPGAs*, pages 16–25. ACM, 2016.

[45] Naoto Sugaya, Masanori Natsui, and Takahiro Hanyu. Context-cased error correction scheme using recurrent neural network for resilient and efficient intra-chip data transmission. In *IEEE 46th Int. Symp. on Multiple-Valued Logic (ISMVL), 2016*, pages 72–77.

[46] S. I. Venieris and C. Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47, May 2016.

[47] Maurice Yang, Mahmoud Faraj, Assem Hussein, and Vincent Gaudet. Efficient hardware realization of convolutional neural networks using intra-kernel regular pruning. *arXiv preprint:1803.05909*, 2018.

[48] Song Yao, Song Han, Kaiyuan Guo, Jianqiao Wangni, and Yu Wang. Hardware-friendly convolutional neural network with even number filter size. *4th Int. Conf. on Learning Representations (ICLR)*, 2016.

[49] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proc. ACM/SIGDA Int. Symp. on FPGAs*, pages 161–170. ACM, 2015.

[50] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[51] Jialiang Zhang and Jing Li. Improving the performance of opencl-based fpga accelerator for convolutional neural network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 25–34. ACM, 2017.