An AI/ML Inference Engine Developed Using AHIR-V2 Tools For Use In AJIT Processor Based Systems-on-Chip

Dual Degree Project Report

Submitted in partial fulfilment of the requirements

for

Dual Degree
(B.Tech + M.Tech in Electrical Engineering
with specialization in Electronic Systems)

by

Aman Dhammani
(Roll No. 18D180002)

Under the guidance of

Prof. Madhav Desai

Department of Electrical Engineering
Indian Institute of Technology Bombay
2023

Declaration

I declare that this written submission represents my ideas in my own words and where others ideas or words have been included, I have adequately cited and referenced the original sources.

I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Aman Dhammani
Roll No. 18D180002
IIT BOMBAY

Date: June 29, 2023

Acknowledgement

I express my gratitude to my guide Prof. Madhav Desai for providing me the opportunity to work on this topic.

Aman Dhammani
Electrical Engineering
IIT Bombay

Abstract

Convolutional Neural Networks form a major part of today's image processing tasks. The continual rise in depth of such networks, coupled with larger intermediate feature maps leads to a requirement of computation in the order of billions of MACs (Multiply and Accumulate) for processing each image. At the same time, such huge tensor sizes and limited memory bandwidth prevent storing of all data on small on-chip buffers. This calls for the need of accelerators with a large number of processing elements designed for efficient reuse the data.

We present a high-performance input-streaming based CNN inference engine operating with small buffer sizes and designed using AHIR-V2, a high level synthesis framework. Using the given design, we demonstrate an end-to-end image segmentation pipeline on a commercially available FPGA, achieving an average performance of over 83 Gigaoperations per second (GOPS), with a peak performance of 96 GOPS when run at a clock frequency of 125MHz. We also integrate a cluster of 8 inference engine with a 32-bit AJIT processor and a Network Interface Card to generate an SoC achieving an average throughput of over 300 GOPS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning is a thriving area in the field of Artificial Intelligence, which involves design and deployment of algorithms that can learn and predict through models trained using given datasets. In recent years, many machine learning techniques, such as Convolution Neural Networks (CNN) [3] and Support Vector Machines (SVM) [4], have shown promise in many application domains, such as image processing and speech recognition [5, 6, 7, 8, 9, 10, 11, 12, 13].

We focus our attention to CNNs, which forms the backbone for most image processing tasks. They are able to capture the image data and modify them through a series of trained filters (kernels) [3]. However, with the increasing depth and complexities of the CNNs, accompanied by large feature maps, it leads to an enormous amounts of computation to process one image, which is beyond the capability of a CPU, which has limited avenes for data-level parallelism. Furthermore, limited on-chip memory necessitates the reuse of the data to reduce memory accesses, requiring specialised architecture with high memory bandwidths and data buffers that can leverage the high levels of parallelism available in such tasks.

Many hardware technologies are available for accelerating machine learning algorithms, which include Graphics Processing Unit (GPU), Field-Programmable Gate Array (FPGA) and Application Specific Integrated Circuits (ASIC). GPUs exploit data-level parallelism to get high performance. Nonetheless, due to its general-purpose nature, it has immense complexity, leading to high power consumption of over 200 W, and hence, is not suitable for embedded applications.

This makes FPGAs and ASICs a promising technology due to their low power consumption, reconfigurability, and real-time processing capability. However, ASICs incur high development cost and therefore, cannot be used for prototyping, making FPGAs the preferred option for development of low-cost ML accelerators [14, 15, 16].

1.1 Design Methodology

The system is designed using C and Aa, a hardware intermediate language [17] capable of describing high-performance digital systems using algorithmic description. We use the AHIR-V2 toolchain [17] for converting modules designed primarily using Aa to generate

the VHDL model of the system. At the same time, the tools produce a couple of binaries using the C testbench written by the user. These testbench binaries can be used to simulate and verify the working of the C model of the hardware, and the generated VHDL design, thereby allowing functional verification before hardware synthesis. The use of algorithmic design and simulation-based verification cuts down the production time significantly as a majority of the functionality in the individual components can be verified and corrected before synthesising the hardware.

The generated design is then instantiated in a top-level VHDL module, which contains other peripheral modules like UART connections, DRAM controller, clock circuitry etc. Appropriate constraint files are set to provide IO functionality for the FPGA card. We use Xilinx Vivado 2019.1 to synthesize, optimise, place and route the system so that it can be deployed on VCU128 FPGA.

Use of the above EDA tools allows the programmers to focus on the algorithmic design of the system, which is then processed to give the most optimal configuration. This helps in cutting down the design and verification time, while simultaneously reducing the code size and complexity, leading to faster and easier debugging [14, 16]. Furthermore, due to implementation of features like pipelining and parallelization, the tools can generate low-level structures having full-rate performance.

1.2 Goals

The project intends to achieve the following goals:

- **Develop and optimize** a high-performance resource-efficient AI/ML inference engine through algorithmic design using AHIR-V2 toolchain
- **Validate and characterize** the accelerator engine on an AJIT-processor based system on chip (SoC) running on VCU128 FPGA, with UNET as the underlying machine learning architecture
- **Demonstrate** the scalability of the accelerator for its application in an inference engine cluster

Chapter 2

Neural Network Architecture Details

There are various CNN architectures used globally for various tasks, like Inception-Net [8], ResNets [10], etc. To develop and characterize our inference engine, we choose UNET as the base architecture. UNET is an CNN-based architecture proposed by Ronneberger et al. for image segmentation[1]. It has been widely used for segmentation tasks, and provides an end-to-end image computation involving several operations used in machine learning, like convolution, padding, maxpooling, upsampling, skip connections, ReLU and sigmoid activations, making it an ideal candidate to deploy on our accelerator. We first describe the UNET architecture, and then the individual operations being used to implement the architecture.

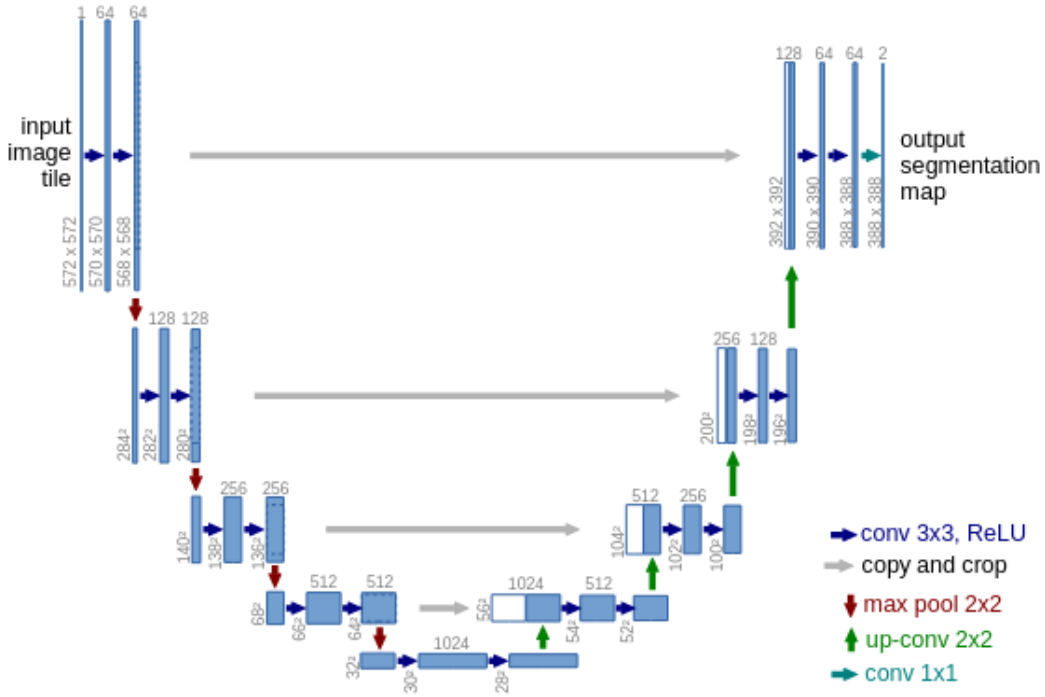

Figure 2.1: Original UNET Architectre (Image Source [1])

The UNET architecture consists of an encoder loop, which will progressively extract the critical information from the image, creating smaller and deeper feature maps. The encoder loop extracts information by performing a series of convolution operation and

maxpooling. The pooling layer helps in reducing the dimensions of the feature maps, helping in encapsulating more and more key features with fewer parameters. The convolution operations are all followed by ReLU activation to maintain non-linerity. There are four encoder loops comprising of two convolutions and a maxpooling layer each. Furthermore, the inputs of the maxpooling layer are stored to be passed on to future layers through skip connections. The left half of Fig 2.1 forms the encoder network.

After the encoding is done, there are a couple of convolution operations to generate the base layer for the decoder. The decoder is designed symmetric to the encoder network. Instead of using pooling layers to downsample the image, it uses transpose convolution (see up-conv in Fig 2.1 operation to upsamples the images to reconstruct the original image with the segmentation data. The grey arrows represent the skip connections from the encoder loops, which help in preserving context identified by the encoder loop. The last convolution reconstructs an image with the same dimensions as the input, giving the segmentation map based on the task at hand.

2.1 Convolution

Convolution operations are extensively used for image processing tasks. Convolution involves sliding the kernel window over the feature map and perform a dot product of the elements covered by the moving window. The tensors are assumed to be 3-dimensional with a dimension each for row, column and channel. The moving window traverses along the rows and columns, and has the same number of channels as the feature maps (input tensor).

We use the following notation scheme to describe convolution. Subscripts `i`,`o`,`k` are used to denote input, output and kernel respectively. We use r, c and chl to represent the parameters row, column and channel for each tensor.
We provide the computational expense of convolution by calculating the number of multiplications (which is the primary compute operation in convolution). Assuming same padding (i.e. input is padded such that output is as large as input), we have $r_o = r_i, c_o = c_i, chl_i = chl_k$ (default). Thus, for a kernel of size $r_k * c_k$, the number of computations to get one output element is $r_k * c_k * chl_k = r_k * c_k * chl_i$. Furthermore, there are $r_o * c_o * chl_o$ such outputs.
This gives the total number of computations for one convolution layer as

$$r_o * c_o * chl_o * r_k * c_k * chl_i$$

For a typical feature map of size 112*112*128, with 128 output channels and 3*3 kernel, this corresponds to a total over 1.8 billion multiplications for a single layer. As we have seen in Fig 2.1, the architecture involves over 18 convolution layers, leading to a computational expense of over 30 billion MAC operations.

We also provide a mathematical description for the convolution operation. A n*n ($\implies r_k = c_k = n$) convolution operation can be defined as:

$$output[r, c, chl] = input[r : r + n - 1, c : c + n - 1, 1 : chl_i] \cdot kernel[chl, 1 : n, 1 : n, 1 : chl_i] \tag{2.1}$$

where $\cdot$ denotes the dot product between two tensors.

Using Equation 2.1, the algorithm for convolution can be described as follows:

```
for co in 1:chl_o do
  for r in 1:r_o do
    for c in 1:c_o do
      tmp = 0
      for r' in 1:r_k d0
        for c' in 1:c_k d0
          for ci in 1:chl_i do
            tmp += input[r+r'-1,c+c'-1,ci]*kernel[co,r',c',ci]
          endfor
        endfor
      endfor
      output[r,c,co] = tmp
    endfor
  endfor
endfor
```

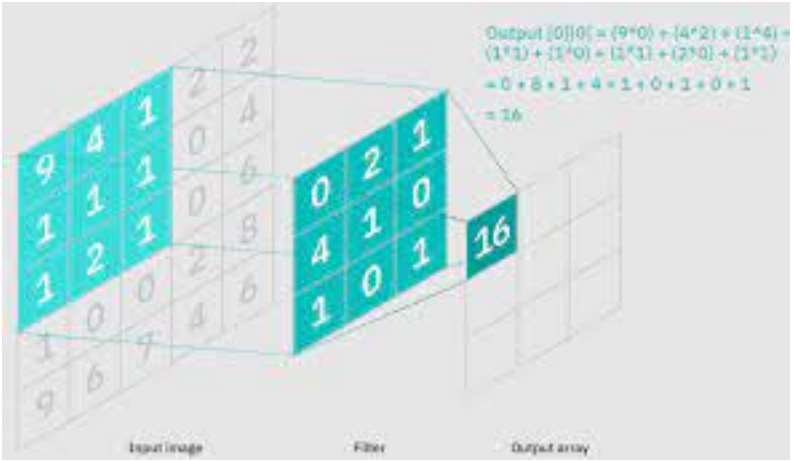

Figure 2.2: Structure Of Convolution Operation (Image Source [2])

In addition to convolution, we have a few other operations that are used in CNNs that account for small portion of computation , the following are supported by the inference engine:

2.2 Pooling

In CNNs, a pooling layer takes a section of the feature map, performs some reduction operation like average, maximum etc. to yield a scalar value. This operation is called pooling. Mathematically, a n*n pooling with a stride of s is given by:

$$Output[r, c, chl] = reduce(Operator, Input[(r-1)s+1 : (r-1)s+n, (c-1)s+1 : (c-1)s+n, chl])$$

where the reduction can happen using any associative operator, like sum, maximum, minimum.

Our engine supports the most common pooling seen, which is maxpooling for n=2 and s=2, which computes the outputs as follows:

$$Output[r, c, chl] = reduce(maximum, Input[2r - 1 : 2r, 2c - 1 : 2c, chl])$$

2.3 Residual and skip connections

Residual and skip connections are an important component of modern deep learning architectures as they help in forwarding information and hence, context over distant layers without passing the weights through non-linear activations, and were introduced in the ResNet architecture [**?**]. The former involves an element-wise summation of two tensors to give the resultant tensor. The latter, on the contrary, concatenates the two tensor maps along a fixed axis (generally the channels) to form the output.

2.4 Transpose convolution

Also known as deconvolution, this operation helps in upsampling of the featrure maps. This is done by adding intermediate zeros between the elements of each feature map to give a much larger feature map,on which a regular convolution operation is performed to give the resulting output tensor.

2.5 Non-linear activation

Non-linear activation functions are applied at the end of each compute layer to improve the model's ability to learn from the dataset. Common activation functions include sigmoid, tanh and Rectified linear unit (ReLU).

Mathematically, we have the following:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

$$ReLU(x) = x * I_{x>0}$$

where, I_a is the indicator variable which takes value 1 if a is true else 0

2.6 Our Architecture

Since the goal of the project is to highlight the efficiency of the engine to accelerate ML tasks, we use a trimmed down version of the UNET architecture [1] as we are not

concerned with the performance of the model itself. Note that this trimming doesn't affect the hardware's performance or its ability to execute the functionality of the original architecture, but reduces the compute workload for demonstration. The modified architecture is described by the pseudocode provided here:

2.6.1 Pseudocode

The following algorithm describes the implementation of UNET (the parameters used in each function calls are not shown here).

```
def UNET:
    T = read_input_tensor()
    K = read_kernels()
    for iter in [1:3] do
        convolve(T,K[2*iter-1], T')
        relu(T')
        convolve(T',K[2*iter],T)
        relu(T)
        R[iter] = store(T)
        pool(R[iter], T)
    endfor

    convolve(T,K[7],T')
    relu(T')
    convolve(T',K[8],T)
    relu(T)

    for iter in [1:3] do
        convolveTranspose(T,K[6+3*iter],T')
        relu(T')
        concatenate(T',R[4-iter],T)
        convolve(T,K[7+3*iter],T')
        relu(T')
        convolve(T,K[8+3*iter],T')
        relu(T')
    endfor
```

```
convolve(T',K[18],T)

sigmoid(T)

send_output_tensor(T)
```

2.6.2 Datatypes Used

Studies have found that fixed point computation for CNNs can achieve similar accuracies with significantly lower logic utilisation, and thereby boasts of higher resource efficiency [15, 16]. Hence, we design the engine for a 8-bit fixed point format for storing all the data. The multiplication gives a 16-bit output, which is then accumulated in 32-bit buffers, which is similar to the implementation done in Google TPU [18]. The scaling at the end is done using a 32-bit multiplier, from which the appropriate 8 bits are selected for write-back into the next stage. Use of fixed-point data simplifies the hardware, allowing faster inference. The position of the decimal and scale valies are software controlled, and appropriate scaling and shifting can be done after each stage to ensure that the format is maintained with minimum loss of accuracy.

Chapter 3

Design

In order to perform inference tasks successfully, the engine must have the ability to perform operations like concatenation, convolution, zero-padding, max-pooling, non-linear activation (relu) and transpose convolution. Fig 3.1 shows the block diagram indicating the modules used and the flow of data within the system. On the left are the set of input modules and the kernel modules. They are responsible for fetching the inputs and the kernels from the memory. They are supported by the readModules which are connected through a deadlock prevention mechanism. The fetched data is forwarded through pipes to the convolveCore, where the bulk of the computation occurs. The partial sums are then accumulated in the accumulator, after which the outputs are generated through scaling and applying activation functions before being written back to the memory using the sendModules. The design and working of individual modules are described in the following sections.

The primary requirement from the engine is that it should be able to perform all the operations listed above with a high throughput. In order to do so, a core unit that can perform multiple parallel MACs is important. Since the core operation of CNNs is convolution, and all other operations are generally done before or after convolution, we cascade the layers to generate hardware that executes other operations as a preprocessing or postprocessing step to the execution of convolution layer. By doing so, we eliminate the need for a memory access after every layer, and hence, the latency of operations improves significantly. We call the combined execution of the preprocesssing, one layer of convolution and postprocessing as a stage. By structuring the engine in this manner, we are able to perform an end-to-end UNET implementation in 18 stages, as opposed to the 38 layers required during training. In addition to the parallelism to generate high throughput, we require minimum data movement across the processing elements to ensure low resource consumption and interconnect delays. So, we use a 384 processing element streaming architecture as described in Section 3.2.1.

An important design consideration while developing the inference engine is the memory sub-system. For UNET, we have observed that the tensor sizes can go over 3MB, and the largest kernel exceeds 2MB in size. It is infeasible to have buffers storing all the information for a given stage. However, we can note that for a 3x3 convolution, an output at row r_i and column c_j is affected only by inputs in rows r_i to r_{i+2} and columns c_j to c_{j+2}. Hence, as we increment the output location, we move to the next input element. This allows us to fetch inputs by streaming them through the pipeline to process the outputs.

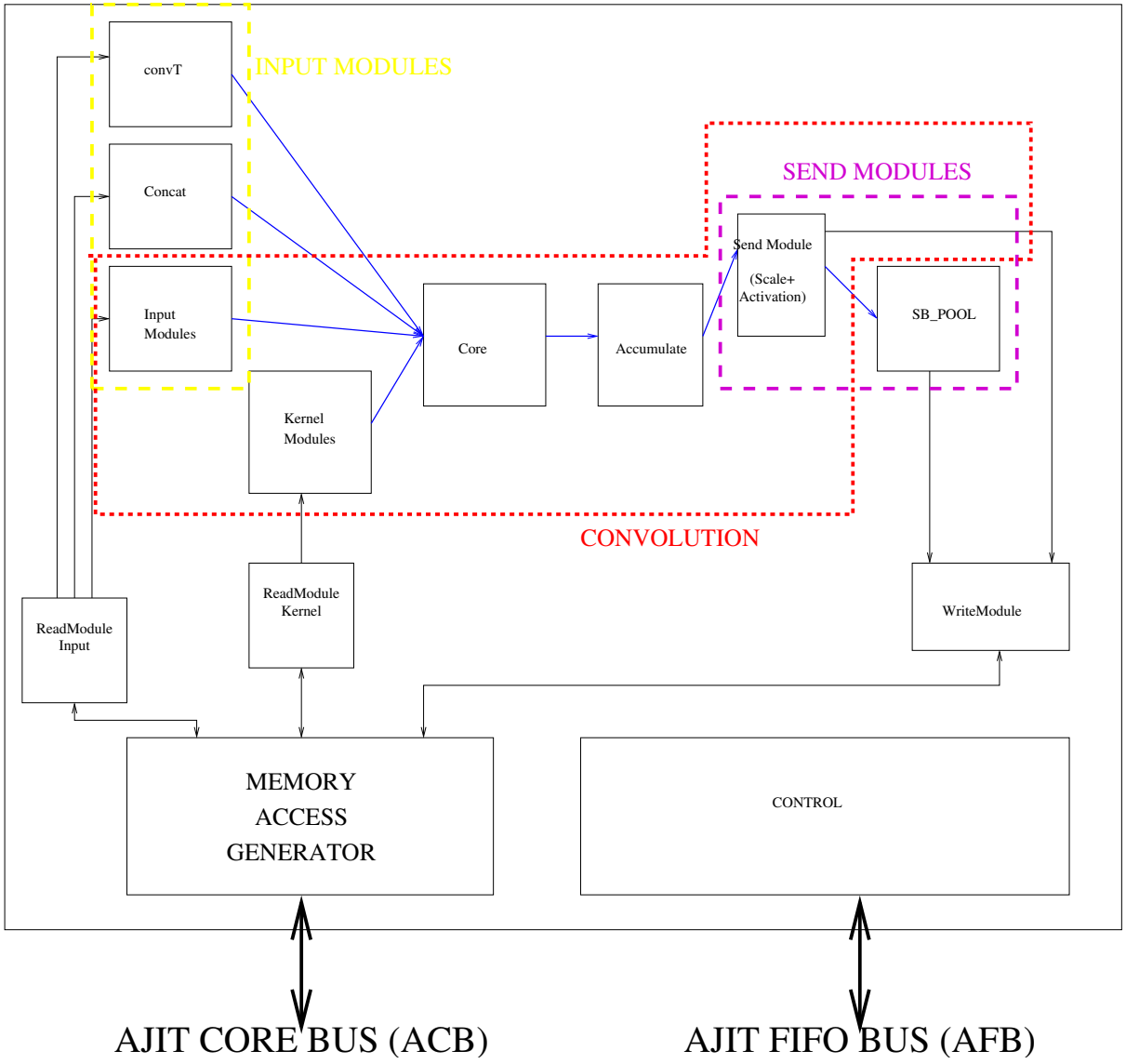

Figure 3.1: Block Diagram For The System Implementation

However, each input is used $r_k * c_k * chl_o$ times throughout the execution, which means that the memory IO requirements for input fetching could go over 200 million accesses for a given stage, with each input being fetched over 4500 times. This could lead to a huge load on the memory bandwidth, and will undoubtedly slow down the system. This calls for the need of a system to reuse the inputs completely before they are discarded. The importance of reuse of data is most noticable for kernel data, which are used as much a sr_*c_i, which can be in excess of 50000 for feature maps of size 224*224 seen in UNET implemented by us. For layers having larger image size like 572*572 used in the first stage original architecture [1], each kernel value is used a staggering 327184, making it extremely necessary to optimise on the kernel reuse.

The most direct way is to buffer the kernels directly. This method is feasible if the size of the kernels are small, however real life kernels can be extremely large. An example is the kernel used in the UNET architecture just before the upsampling block. This kernel has $3*3*1024*1024$ elements, which will require a storage space of 9MB using 8-bit data.

To mitigate this issue, the kernels are partitioned into units that can be stored on the

on-chip FIFOs. The partitioning is done to accommodate as many output channels as possible in the buffers, thereby minimising the number of partitions. The inputs have to be streamed once for every kernel partition. Since the number of times each input is fetched is equal to the number of partitions, it is imperative to ensure that the number of partitions are kept as low as possible to improve system performance. Currently, 768 KB of kernel buffer is provided, which keeps the maximum number of partitions as 3 for the architecture deployed.

The design involves two aspect - the critical component involving the convolution core, internal buffer management and memory IO, and the non-critical components for auxiliary tasks like concatenation, maxpooling, padding and applying non-linear activations.

3.1 Algorithm

Rewriting Algorithm 1, we get the following algorithm that is implemented by the two modules (blue denotes convolveCore specific tasks, red denotes accumulator tasks):

```
1  // Two output rows at a time
2  for r in 1:2:r_o do
3    for c in 1:c_o do
4      // 8 output channels simultaneously
5      for co in 1:8:chl_o do
6        partial_sum[2,8] = 0
7        for c' in 1:c_k do
8          // 8 input channels at a time
9          for ci in 1:8:chl_i do
10
11           // The below part happens in one loop of the core
12           // Hence we replace for with for_unrolled which signifies
13           // that the loop is unrolled over the range of its iterators
14
15           // Temp variable for 384 multiplications
16           // which are accumulated and reduced to 16 partial sums
17           tmp[2,8,8,3] = 0
18           tmp_reduced[2,8] = 0
19           for_unrolled co' in co:co+7 do
20             for_unrolled r' in 1:r_k do
21               for_unrolled ci' in ci:ci+7 do
```

```
22                  tmp[1,co',ci',r'] = input[r+r'-1,c+c'-1,ci']*kernel[co',r',c',ci']
23                  tmp[2,co',ci',r'] = input[r+r',c+c'-1,ci']*kernel[co',r',c',ci']
24                end_unroll
25              end_unroll
26              tmp_reducei[1,co'] = sum(tmp[1,co',:,:])
27              tmp_reducei[2,co'] = sum(tmp[2,co',:,:])
28            end_unroll
29            endfor
30            sendToAccumulator(tmp_reduce)
31            receiveFromConvolveCore(tmp_reduce)
32            partial_sum += tmp_reduce // Element wise sum
33          endfor
34        endfor
35      endfor
36    endfor
37  endfor
```

3.2 Critical components

3.2.1 convolveCore

Organisation

The convolveCore is the most critical component in the accelerator. The purpose of the module is to ensure that a large number of computations can be carried out simultaneously to facilitate the acceleration we desire. In order to do so, the core unit is designed as a 2*8*8*3 multiplier system. This provides a theoretical peak of 768 operations per clock cycle. The organisation of the multiplier array was chosen to allow for two rows of ouputs processed 8 channels at a time, with support of kernels upto 3 rows, and inputs received 8 channels at a time. The use of 8 input/output channels was motivated by the use of 8-bit datatype, allowing us to process an write-back whole word at a time, thereby improving memory performance. Also, two rows of outputs are delivered simultaneously, as this facilitates easier maxpooling, as the two outputs can be compared immediately to get the pooled result.

The processing elements can be viewed as 6 logical sections of 8*8 multipliers arranged as shown in Fig 3.2, and represents the unrolling obtained in line 20,22 and 23 of Algorithm 1. The inputs and kernels are multiplexed such that the first three sections compute the partial products for the first output row, while the next three sections produce the partial products for the second output row. Since convolution uses a sliding kernel window, the kernel remains the same across each row, while the input is shifted by a single

row. This implies that the second and third inputs are utilised by two sections each.

Each logical section can be broken down into an 8*8 processing element grid, where there are 8 sections corresponding to the 8 output rows. All of them receive separate kernel values but the same inputs. Each sebsequent block receives 64-bit values, which are sliced into 8 8-bit values and a dot product over them is computed. This corresponds to the for_unrolled loop in the algorithm described in line 21 of Algorithm 1. The outer 8 sections each represent a unit from the unrolling of line 19 of the same algorithm.

Data Buffering And Reuse

During the processing of one stage, the core module iterates over the above computation multiple times as mentioned in lines 2,3,5,6 and 8 of Algorithm 1. This amounts to a total of $\frac{r_o * c_o * chl_o * chl_i * c_k}{128}$ iterations. Furthermore, in order to release the partial sum from the storage buffers immediately, each output must be computed before the next one is started, which leads to lines 6 and 8 being executed before the outer loops. Thus, the flow of inputs through the core module involves passing all the channels corresponding to the given row and column before moving to the next column. After a column c is used for computaion, the next $c_k - 1$ columns are sent, after which again c+1 to $c + c_k - 1$ are sent, and so on. For $c_k = 3$, all columns are used 3 times before the next row is sent in. Since the outputs are processed 2 rows at a time, the input is sent at intervals of two rows, i.e. the first input pipe (core_ip1) will receive input rows 1,3,5, and so on, while the second input pipe (core_ip2) will receive input rows 2,4,6 etc .

Looking at the flow of kernels, they have to be sent corresponding to the inputs they multiply with, and so, they are sent channel by chaanel, followed by columns. Since all rows of the kernel are sent simultaneously (they are at most 3), the kernels iterate over the output channels, for each of which the same input needs to be streamed again.

Thus, we observe that each input is being used c_k times for different output columns, r_k times for different output rows, and chl_o times for different output channels, and the kernel is reused $c_o * r_o$ times as it slides or each output element of a feature map. Out of which, the architecture reuses the input 1.5 times and kernel twice. If there are no mechanisms for the inputs and kernels, we will require a very high memory bandwidth of $\frac{48}{1.5}$ accesses per clock for inputs, and $\frac{48}{2}$ accesses for inptus for kernels, leading to 56 memory accesses every clock cycle) to maintain full rate of execution.

As the memory bandwidth is limited to 1 access per clodk cycle, we provide the core with pipes that serve as buffers for reuse of data. There are 4 pipes of depth 256 (total of 8KB buffer) for buffering inputs, and are designed to store 3 columns of input data. This improves the input reuse by a factor of $3 * chl_o$ by reusing the inputs across all output channels and kernel columns, bringing the average input bandwidth requirement down significantly to around $\frac{48}{4.5 * chl_o}$, which in the range of 0.02 - 0.2 accesses every clock cycle as chl_o ranges between 64 and 512.

Observing that the kernel is reused a few thousand times, it makes sense to store the kernels completely on chip as the added storage is justified by the immense savings in memory accesses. To facilitate that, 24 pipes of depth 4096 (32KB each for a total of

768KB) are used to provide 3 rows of kernelsand 8 output channels at a time. Thus, we ensure full reuse of kernels and bring down the average kernel bandwidth requirement negligible. In some cases, the kernel size may exceed the buffer sizes. In such scenarios, the output is partitioned based on the number of channels, and a few channels are computed at a given point in time. Thus, only the kernels corresponding to the output channels are buffered. The partitioning is done in a way to ensure maximum use of the kernel pipes, which will lead to the minimization of the number of times the input is fetched, since all the inputs are fetched as many times as the number of kernel partitions.

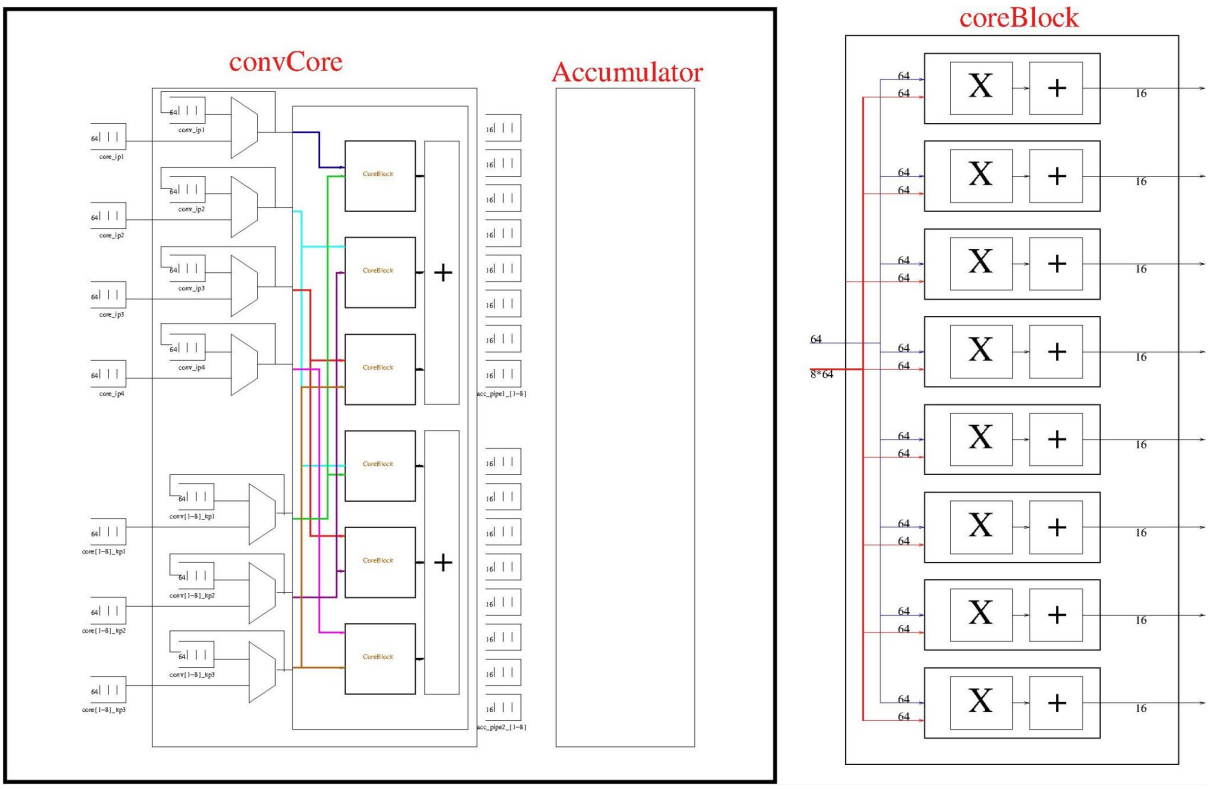

Figure 3.2: Organisation of the critical modules - convolveCore and accumulator

3.2.2 Accumulator

The accumulator is implemented independent of the core unit. This is done to improve the performance of the system as each iteration of the core can run in one clock cycle. The accumulator reads the values and accumulates till count equals the parameter acc_count, which corresponds to the number of product of the number of channels and columns of the input required to get a output. After that, the counter and the accumulated value resets and waits for the next set of partial sums to produce the next output. The accumulator unit is capable of handling 16 sets of partial sums every clock cycle.

The accumulator reads 16 output values across 2 rows and 8 output channels, and maintains partial sums that are 32b wide, which is the same used in TPUs [18]. On completion of every accumulation, the data is forwarded to the sendModule/sendModule8/sendPool through sendPipes for further processing and writeback to the memory.

3.3 Non-critical components

Convolution module is a collection of multiple smaller modules, and has the ability to perform zero-padding, maxpool, concatenation, transpose convolution and non-linear activation in addition to its core functionality. This can be done as all the above operations are linear in their mapping from input to output, and hence any stream can be modified insitu to get the modified stream. This facilitates the use of specialised input/output modules to achieve the above operations.

Thus, the overall scheme reduces to the design of an all-purpose core which can be integrated with each of the input/output configurations to get the desired functionality. The convolution module can perform 8-bit multiplication on int8 data, with 32 bit accumulation. Accumulator is kept as a separate entity than the multiplier core to facilitate full-rate execution as this leads to no data dependency in the multiplication unit.

3.3.1 Input Modules

In order to reduce the number of memory accesses and provide a steady stream of data to the convolution core, a series of modules were implemented to deliver the inputs at the best possible rate based on the operation to be performed.

This modules reads the input tensor from the memory, and sends it to the core for processing. The modules are designed to fetch two input rows at a time. This is done due to two reasons:

1. Keeping the number of rows to two allows us to fetch the input data only once per kernel partition in a layer instead of two times required by the convolveCore, leading to full reuse. As we increment the number of rows by two, we forward the same data twice to the convolveCore using the singleFetch module described below.

2. Since the output produces two rows at a time, having two input rows provides an opportunity to cascade multiple engines to eliminate intermediate memory accesses.

All the input modules have support for padding with zeros by sending in zeros at the desired positions. This enables us to prevent restructuring of the tensor to enable zero-padding. They call a two concurrent submodules to fetch one row each - the first module handles the odd numbered rows, while the second handles the even numbered rows. The modules ensure that the zeros created due to padding are transmitted to the singleFetch module. Fig 3.3 represents the organisation of the input modules in the system.

Single Fetch

As discussed in the convolveCore section, each input has to be streamed twice for correct operation. However, using the singleFetch module, we provide that functionality through a single read for each input. This module reads the first two input rows and buffers them in the FIFOs for the core. At this point, the core blocks due to unavailability of data in the third and fourth input FIFOs. At this point, the module sends the next row into the FIFO 1 and 3, and the subsequent one to FIFOs 2 and 4. This way, after an initial delay, it gives the core an illusion that all four rows are available simultaneously, while in reality the last two FIFOs lag the first two by $c_i * chl_i$ elements, which is then corrected when

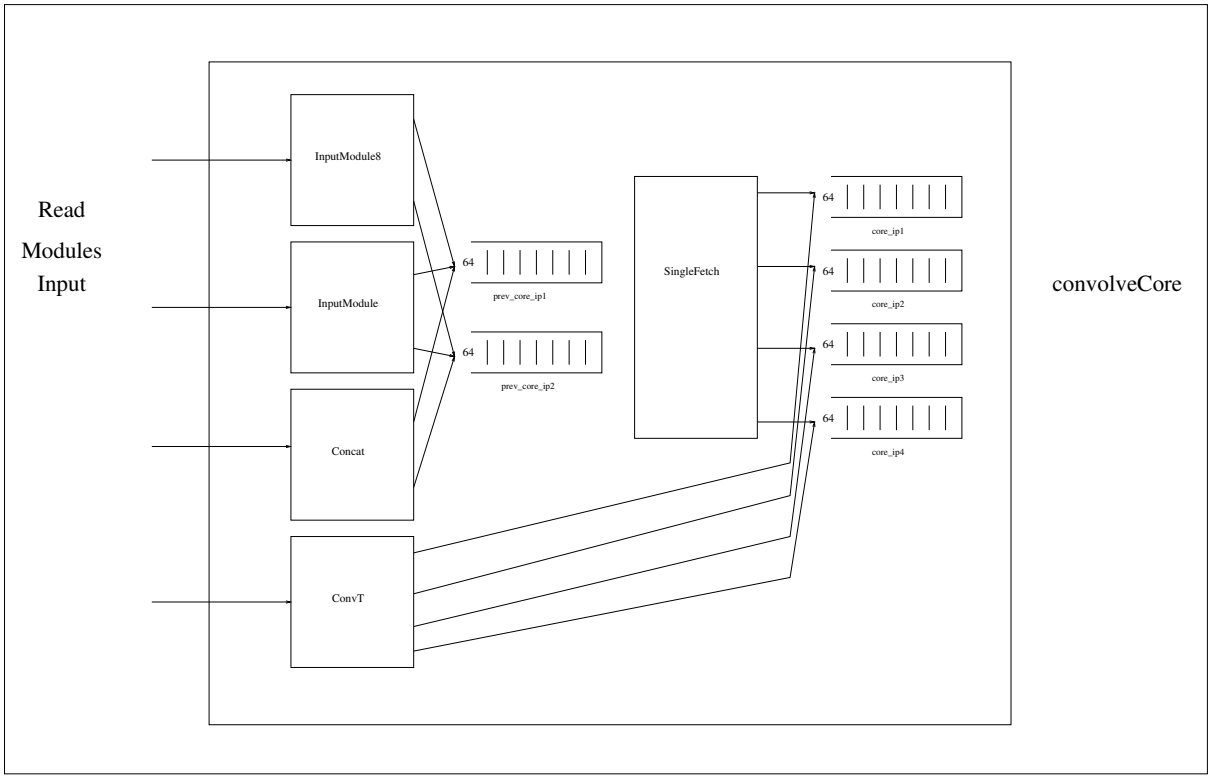

Figure 3.3: Organisation of the input modules

the last row is forwarded only to the third and fourth FIFOs, thereby balancing all inputs.

This way, the number of times each input is fetched kept to a bare minimum of 1, while allowing the core to function at full rate. However, this does require an additional storage to buffer two rows of inputs, which can go upto $2 * 224 * 64$ Bytes = 28KB. Since inputs can be much larger, a total of 64KB is provided for row buffering to execute other architectures.

Unpacking

One of the other functionality of the input modules is to unpack of the input, such that each element is channel-aligned, i.e. the first channel is always at the start of the word. This ensures correct operation when the number of input channels is not a multiple of 8. These are useful at the input and output stages of the CNN where the tensors correspond to real-world images and hence have 3 channels.

Transpose Convolution

This module is designed to replace convTranspose operation. Transpose convolution involves the following operations:

1. Writing the tensor with zeros at appropriate spacing
2. Depadding the extra layers in the tensor
3. Performing convolution operation on the resultant tensor

This input modules does the first two steps, which reduces the operands to a form that can be processed by the convCore module. To optimize the performance, the two steps are done simultaneously by performing depadding by not padding with zeros at locations which will eventually get depadded. The padding and depadding happens by addition of zeros at the appropriate rows and between adjacent columns. The module reads in the data from the memory, but interleaves it with words having all bits zero to get the desired dilated tensor, which is then forwarded to the compute engine for processing. This module also assumes that the number of input channels is a multiple of 8, and hence, the fetched word is forwarded as it is without unpacking.

Concatenation

The input modules can also help in concatenation along the channels to implement skip connections, which is executed by alternately reading columns from the two tensors, and sending the data stream to the core. This is done by using two counters for the channels to determine when to switch from one tensor to the next. This interleaving results in the concatenation of the two tensors.

Since concatenation occurs only during the decoding loop, it has inputs and output which have channels that are multiples of 8. This permits the module to read whole words from the memory at a given point in time and send it as it is as the output. Hence, the module is optimised to directly forward whole words from one memory address to another.

3.3.2 Kernel Modules

This set of modules are responsible for fetching the kernel data from the memory, and sending it to the core. Like the input module, it can unpack the data channel-wise to ensure channel-alignment, ensuring correct operation when the number of input channels is not a multiple of 8. The kernel data is sent through three pipes, one for each row.

The kernel module makes calls to submodules to fetch kernels for different rows. Since r_k is at most three, the submodules need not fetch alternate rows. The fetch is done channel by channel, and is sent to pipes core$< i >$_kp$< j >$, where $< i >$ is the core number (corresponds to chl%8), and $< j >$ is the row number, i.e. different for each submodule.

Minimum latency and buffer requirements

The kernel module sends the data to the core using 24 pipes, 3 each for 8 output channels processed together. Since the convolution core requires all 8 cores to have valid kernels, there must be enough space in each pipe to store the data corresponding to one input channel, and hence the `INTERMEDIATE_PIPE_DEPTH` is set to 512, which is more than enough for all requirements. This leads to a total buffer requirement of 96 KB for storing intermediate kernels, and since the input pipes are also parameterised by the same depth, they are supported by a buffer of size 16 KB.

As this module transmits kernel values for 8 channels in parallel, we get loop unrolling with 8 loops unrolled. This also allows the convolution module to start processing imme-

diately as it has the kernels required for first computation ready in minimal time. This reduces the startup latency significantly.

3.3.3 Output Modules

The output modules receives the data from the accumulator, and writes them back into the memory after applying the required operations. If the number of output channels is not a multiple of 8, it packs the data into words before writing it into memory. All the send modules are equipped to perform non-linear activation on the output data which is received from the accumulator through two sets of eight FIFOs each 32bit wide. The organisation of the modules in the system is shown in Fig 3.4.

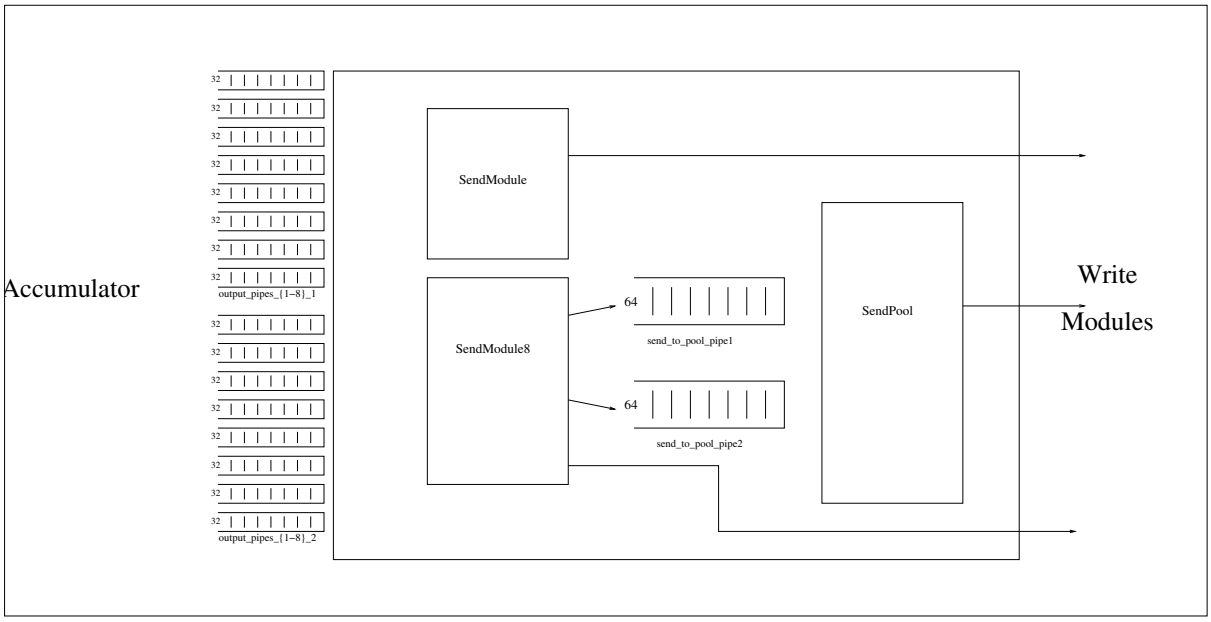

Figure 3.4: Organisation of the output modules

Scaling

The received data is sent to the scaler module, which translates the data by multiplying with a 32-bit scale value which is fixed for a given stage of the network. When the multiplication is done, the data is sent back to the calling module for shifting the data to obtain the final value. This way, a high accuracy can be obtained by minimising the loss of information. During a read from the accumulator, 16 outputs are fetched, but only 1 scaler is used as the accumulator provides data once every $\frac{3n}{8} + 1$ cycles, where n is the number of input channels. Thus, if the number of input channels is 128, writeback happens once every 49 cycles, and hence the multiplier associated with the scaler module can be shared across all outputs.

Non-linear activations

The module receives the data and performs ReLU if the non-linearity flag for ReLU is set. The operation is performed on the accumulated outputs before the scaling is done. The data is then scaled and truncated before being packed into blocks of 8 elements, which is then written back to the memory. The processing is done two rows at a time.

Maxpooling

The outut modules also perform maxpooling after convolution. The pooling operation is done by two modules, the first of which receives the data and performs scaling and shifting. This completes the convolution part and the intermediate result can be stored if needed. The second module receives the scaled data of two rows and two columns, and forwards the maximum of the four values for writeback. Also, since the data is received channel by channel, a small storage of size $chl_o < 512B$ is needed to buffer the partial comparison products. A 2KB buffer is provided for the same.

3.4 Auxiliary components

In addition to the the above modules, we have some auxiliary components that help in the execution of the system.

3.4.1 Read and Write Modules

These modules are an interface between the functional pipelines and the memoryModule, and operate by providing address and/or data along with the appropriate bitmask to the memoryModule for read/write to memory, and delivering the read data back to the calling module in case of a load from memory.

Although There are no cyclic dependencies amongst the core modules, they have one shared resource - the memory which is accessible via the memoryModule. This creates a potential for deadlock as the input module may be blocked by the pipes to the core. This in turn blocks the readModule and hence, the return of data from memory, therefore blocking the memory, which will prevent writeback from the send modules. This will eventually choke the accumulator and the core as the data is still pumped into the send modules but not cleared. In the end, all modules block resulting in a deadlock. Any deadlocks arising due to blocking of shared resource is mitigated in the read modules using the mechanism shown in Fig 3.5.

The mechanism in the figure works on the fact that the number of live iterations of readModule1 will be bound by the pipeline depth 7, and hence, there can be at most 7 entries in the PIPE, and hence, it will never block. Thus, every call to the memory by the readModule1 is eventually returned, thereby preventing any resource contention leading to deadlocks.

It may be noted that there is no need to have a deadlock mechanism for writeModules, since any call to the writeModule eventually returns as the output of the writeModule is not forwarded to a pipe, and hence, is never blocked.

3.4.2 Memory Module

The module forms the interface between the engine and the external memory. It receives read/write requests from other modules through their respective read/write modules. It then transmits the 110-bit request to the ahir core bus (ACB), which relays it to the memory controller for memory access. The memory controller sends back a 65-bit response, which includes a 1-bit error flag and 64-bit data, which is forwarded back to

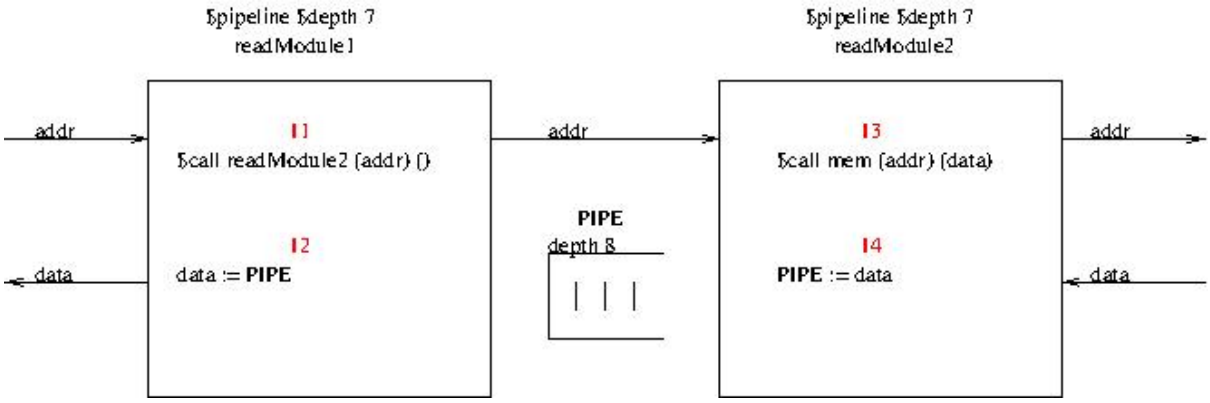

Figure 3.5: Implementation Of Deadlock Free Read Module

the calling module. The memory module provides abstracts the system design from the memory interface. Using an ACB interface, the internal working of the engine just needs to provide the read/write flag, data, address and the bytemask, and wait for the response without having to worry about the actual memory (BRAM/DRAM etc) being used, which can be handled by an external memory controller.

3.4.3 Interface To Access The Engine

The engine is controlled by two ever-running daemons. The first amongst them is the accelerator_control_daemon, which resets the registers and starts the worker daemon for operation. It also receives the configuration information from the processor/calling module through the AFB interface, using which it programs the registers.

The second daemon is the accelerator_worker_daemon, which polls on the control register R0 until the bits 0 and 2 are set by the calling module. When the condition is met, it reads the other registers, and makes a call to the convolution engine via the command:

```
$call convolutionAll (
    ro co ri ci cho chi rk ck
    in1_base_addr in2_base_addr kernel_base_addr out_base_addr1 out_base_addr2
    scale_val shift_val pad pool concat act
    ) ()
```

where,

- ro, co, ri, ci, cho, chi, rk and ck are 16-bit values providing the dimensions of all the tensors and kernels (notation same as in Chapter 2)
- in1_base_addr, in2_base_addr, kernel_base_addr, out_base_addr1, out_base_addr2 are the base address for the location of upto two input and output tensors, and one kernel. During pooling, out_base_addr2 is used to configure the address to which the pooled outputs are sent. Two input base addresses are used during concatenation (skip connections)

- scale_val and shift_val are 32-bit and 16-bit value specifying the scale value and the shift value
- pad specifies the amount of zero-padding required on the input before convolution
- pool, concat, act are 8-bit control flags that instruct the engine to use pooling, concatenation and/or non-linear activation

There is also an interrupt module, which will generate an interrupt signal whenever the engine completes its execution.

3.4.4 Register File Format

The accelerator engine can communicate with the external world through 16 32-bit registers as follows, which can be used to exchange the following information:

- Register 0 : Serves as the control for the accelerator, and to see status flags
- Register 1 : Stores the value of the number of output rows (31 downto 16) and number of output columns (15 downto 0)
- Register 2 : Stores the value of the number of input rows (31 downto 16) and number of input columns (15 downto 0)
- Register 3 : Stores the value of the number of output channels (31 downto 16) and number of input channels (15 downto 0)
- Register 4 : Stores the value of the number of kernel rows (31 downto 16) and number of kernel columns (15 downto 0)
- Register 5 : Stores the value of shift_val for rescaling (31 downto 16) and padding (15 downto 0)
- Register 6 : Stores the control for concatenation (23 downto 16), pool (15 downto 8) and activation (7 downto 0)
- Register 7 : Stores the base address for the first input
- Register 8 : Stores the base address for the second input, will be unused if the MSB is 0
- Register 9 : Stores the base address for the primary output
- Register 10 : Stores the base address for the secondary output, used when both output and the pooled output are to be stored
- Register 11 : Stores the base address for the kernel
- Register 12 : Stores the scale value (31 downto 0) for scaling the outputs before shifting and write-back

Registers 13-15 are unused, but can be modified for other parameters and/or debug purposes.

Register 0 has the following bits used:

- Bit 0 : accl_enable - enables the engine, can be set/reset from outside the engine

- Bit 1 : interrupt_enable - enables the interrupt flag, can be set/reset by calling module
- Bit 2 : start_cmd - set by calling module, instructs the accelerator to begin execution
- Bit 3 : cmd_complete - set by the accelerator to indicate that the execution is complete, reset by processor when signalling next command
- Bit 4 : accl_done - set by accelerator to signal that the execution is complete and interrupt is to be generated

3.5 Sample dataflow

We show a small example of how the data flows through the engine. Assume a small 2*2 input tensor with 8 channels being convolved to give a 2*2 output tensor with 8 channels. The padding is 1, with no pooling, concatenation or transpose convolution taking place. Assume that the data is stored numerically at each double-word. Thus, the input will be a 32-byte (2*2*8) tensor with value i at double-word i, i.e. [1,2;3,4] using the standard MATLAB notation for arrays. The kernel will be 576 bytes (8*3*3*8) large.

We denote the flow of information in the FIFOs from left to right, with left being the earliest entry. The input modules will generate the following values into the pipes by incorporating padding which transforms the input to [0,0,0,0;0,1,2,0;0,3,4,0;0,0,0,0]:

- prev_core_ip1 := 0 , 0 , 0 , 0 ; 0 , 3 , 4 , 0
- prev_core_ip2 := 0 , 1 , 2 , 0 ; 0 , 0 , 0 , 0

These values are then forwarded by the singleFetch module to the core will be:

- core_ip1 := 0 , 0 , 0 , 0 ; - , - , - , -
- core_ip2 := 0 , 1 , 2 , 0 ; - , - , - , -
- core_ip3 := - , - , - , - ; 0 , 3 , 4 , 0
- core_ip4 := - , - , - , - ; 0 , 0 , 0 , 0

where '-' indicates that no data was sent to the pipe in that iteration of the module

Similarly, kernel module sends the following data for i ranging from 1 to 8:

- core< i >_kp1 := 9*(i-1)+1 , 9*(i-1)+2 , 9*(i-1)+3
- core< i >_kp2 := 9*(i-1)+4 , 9*(i-1)+5 , 9*(i-1)+6
- core< i >_kp3 := 9*(i-1)+7 , 9*(i-1)+8 , 9*(i-1)+9

Based on this, the convolution module generates the following streams in their local buffers

- conv_ip1 := 0 , 0 , 0 , 0 , 0 , 0
- conv_ip2 := 0 , 1 , 2 , 1 , 2 , 0
- conv_ip3 := 0 , 3 , 4 , 3 , 4 , 0

- conv_ip4 := 0 , 0 , 0 , 0 , 0 , 0
- conv< i >_kp1 := 9*(i-1)+1 , 9*(i-1)+2 , 9*(i-1)+3 , 9*(i-1)+1 , 9*(i-1)+2 , 9*(i-1)+3
- conv< i >_kp2 := 9*(i-1)+4 , 9*(i-1)+5 , 9*(i-1)+6 , 9*(i-1)+4 , 9*(i-1)+5 , 9*(i-1)+6
- conv< i >_kp3 := 9*(i-1)+7 , 9*(i-1)+8 , 9*(i-1)+9 , 9*(i-1)+7 , 9*(i-1)+8 , 9*(i-1)+9

From the above, we see that the input and kernels are always matched appropriately. This simplified example shows how the modules use the buffers to generate the required functionality by fetching only the data only once from the memory. Also note that the number of inputs and kernels received from the corresponding fetch pipes are not the same (4 and 3 respectively), but the reuse techniques ensures that the circular FIFOs are appropriately filled and emptied to have exactly the correct number of inputs and kernels.

3.6 Validation and Performance

The inference engine was synthesized using Vivado 2019.1 and run on a Xilinx VCU128 FPGA at a clock frequency of 125MHz with the external memory modeled using BRAM,. With a theoretical peak performance of 96 GOPS and logic cells resource consumption of 102K LUTs, we get an average of 0.94 GOPS/KLUTs performance. The end-to-end segmentation using UNET architecture on the engine takes a total of 94 million clock cycles, taking a total time of 0.75s for processing of one image, giving an average performance of over 85 GOPS.

We can verify the full rate of execution from the stage by stage breakdown shown in Table 3.1, where resource utilization of about 99.75% (383 MACs/clock) have been observed. We see that the average compute utilisation of the engine is close to 90%, indicating the efficient use of compute resources. Furthermore, the aggressive reuse of data keeps the average memory utilisation under 10%, which means that the engine is bottlenecked by the compute resources and not the memory. The results indicate an average compute-to-memory ratio of 873 operations/byte (1 MAC = 2 ops). This implies that the engine can be easily scaled to larger engine clusters sharing the same memory channel without significant degradation in preformance. We study this in Section 4.3.

3.7 Comparison With Previous FPGA Accelerators

Table 3.2 shows a cmparative analysis of the performance of our implementation with some of the reported literature for fixed-point CNN acceleration.

Stage	Type of operation	Time taken (*10^6 clock cycles)	Computation (*10^6 MAC operations)	Memory Accesses (*10^6 accesses)	Compute Resource Utilization (%)	Memory Resource Utilization (%)
1	Convolution*	1.6092	86.70	0.4204	14.03	26.13
2	Convolution + Pool#	4.8282	1850	1.008	99.78	20.88
3	Convolution	2.4266	925	0.3103	99.43	12.79
4	Convolution + Pool#	4.8531	1850	0.5202	99.27	10.72
5	Convolution	2.4807	925	0.1874	97.10	7.55
6	Convolution + Pool#	4.9614	1850	0.3246	97.10	6.54
7	Convolution	2.6726	925	0.2478	90.13	9.27
8	Convolution	5.4155	1850	0.5458	88.96	10.00
9	Transpose convolution	6.5496	1644	0.2161	65.37	3.30
10	Convolution#	9.8979	3700	0.6492	97.34	6.55
11	Convolution	4.9614	1850	0.2744	97.11	5.53
12	Transpose convolution	6.4545	1644	0.3174	66.33	4.92
13	Convolution#	9.7061	3700	0.6390	99.27	6.58
14	Convolution	4.8531	1850	0.4198	99.27	8.65
15	Transpose convolution	6.4306	1644	0.6062	66.58	9.43
16	Convolution#	9.6577	3700	1.2134	99.77	12.56
17	Convolution	4.8289	1850	0.8074	99.77	16.72
18	Convolution*	1.2850	86.70	0.4204	17.57	32.71
-	End-to-end	93.872	31930	9.1390	88.58	9.74

Note: All the convolution are succeeded by ReLU, hence it is not mentioned in the table

* - Input/output has 3 channels and are not aligned with memory, hence engine needs to align them and pad with zeros. Can work at a maximum utilisation of 3/8

#- Includes memory IO for skip connections

Table 3.1: Stage by stage memory and compute performance of the accelerator engine

Architecture	Board	Model	Design Methodology	LUTs (K)	Throughput (GOPS)	DSP Slices	Frequency (MHz)	Throughput / Logic-freq (GOPS / KLUT GHz)	Throughput / DSP-freq (GOPS / KLUT GHz)
[19]	Stratix 10	VGG16	RTL	469	1604	8216	300	11.4	0.65
[20]	VX980T	VGG16	RTL	335	1000	3395	150	19.9	1.96
[19]	Arria 10	VGG16	RTL	208	968	3036	240	19.4	1.33
[21]	VU9P	ResNet-50 (16b)	HLS	692	1672	5632	180	13.42	1.65
[20]	VX980T	ResNet-101	RTL	480	600	3121	100	12.5	1.9
[22]	VX690T	VGG16	RTL	232	761	1027	200	16.40	3.70
[23]	ZC7045	UNET (16b)	HLS	86	107	640	200	6.22	0.84
[24]	Arria 10	UNET	RTL	250	1578	1688	200	31.56	0.93
Our Work	VCU128	UNET	HLS	107	85	423	125	6.35	1.61

Table 3.2: Performance evaluation with state of the art FPGA implementations

Chapter 4

Integration into a System-on-chip

We validate an characterize the performance of the accelerator by integrating it into a System-on-chip (SoC) using AJIT Processor, which provides the accelerator with the commands to operate, and reads back the data. The whole system is also connected to a Network Interface Card, which provides high-speed IO for quick loading of images into the memory. The architecture of the system is shown in Fig 4.1.

The system consists of 32bit wide AJIT processor at its core. The processor has an ACB interface through which it interacts with memory and other modules. The AFB interface linked to the Network Interface Controller (NIC) and the 8-engine acclerator cluster is used to transfer configuration information from the processor and relay back status flags to it. The modules are also provided with an ACB interface through which they can access the memory. The memory subsystem consists of a DRAM controller connected to an ACB to UI conversion protocol, which translates memory requests in the ACB bus to DRAM requests, and vice versa for the responses. In addition to the above, the processor has a couple of UART interfaces with which it can communicate with the outside world. Also, the NIC has an interface to connect with the Xilinx MAC IP, using which it can communicate to the host machine via Ethernet, currently configured to run at 100Mbps.

4.0.1 Performance and validation

The utilisation statistics for different components designed is present in Table **??**.

4.0.2 Processor Code

The pseudocode for the processors code is as follows:

```
1 initialiseMemorySpaces()
2 initialiseNicQueues()
3 fetchKernelsThroughEthernet()
4
5 while(1) do
```

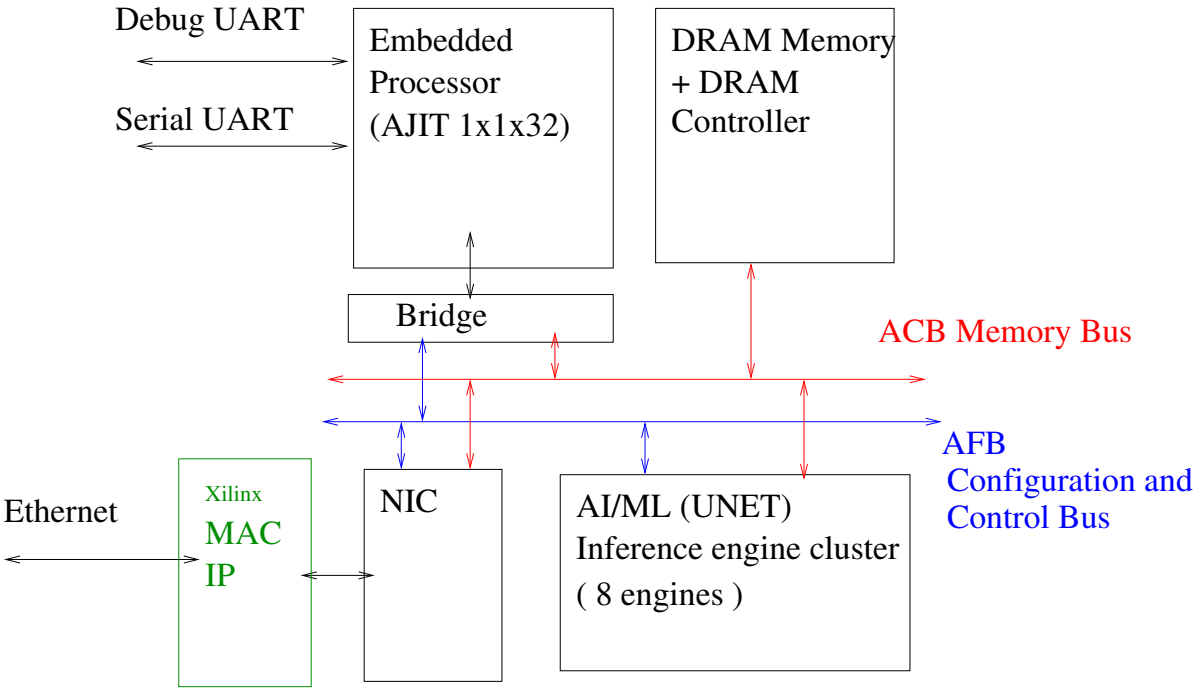

Figure 4.1: Block Diagram For SoC

```
    fetchInputsThroughEthernet()

    initliaseAcceleratorStates()

    while (all accelerators not done) do
      for engine in list_of_engines do
        convolve(engine,stage)
        updateState(engine,stage)
      end for
    end while
    sendOutputsThroughEthernet()
end while
```

The above pseudocode explains the working of the processor code. It first initialise memory spaces required by all modules. These include the queues required by NIC, the kernel addresses for the accelerators and the memory spaces for receiving and storing tensors during computation by the engines. After the memory spaces are initialised, all the queues for NIC are configured and initialised, and the Network Interface Controller starts executing. The kernels are then fetched in through Ethernet, which is then followed by the input tensors for all the engines. With the setup ready, NIC is turned off and the engines are allowed to execute on the data stage by stage. When all the acclerators are done with their computation, the output data is sent to the host PC through Ethernet, where the output is verified for correctness.

The above is a simplified driver to demonstrate the functionality of using multiple accelerator engines inside the system using polling. The processor and the accelerators both support the use of interrupts. The Interrupt Service Routine corresponding to the accelerator interrupts can be suitably confiured to allow the processor to be used for other processes, while the engines work on the compute-intensive AI/ML tasks, the completion of which can be signalled to the processor, at which the processor interrupts and schedules the next task to the accelerator before resuming its own execution.

4.1 Processor NIC Interface

The processor initialises the queues for the NIC and starts the NIC. When NIC recives data from MAC it pops the free_queue which has empty packet buffers address stored in it and writes the packet to that buffer. After successfully writting the packet NIC pushes the address of buffer to rx_queue, any buffer address in rx_queue indicates that this is the received packet and needs some action by processor. While this thing is going on processor keeps polling the rx_queue, as soon as it gets any data over there it reads it and takes decision. In this setup processor is expected to store the packet at some memory location which is shared with acclerator through its registers. After this packet storage is complete processor pushes this address to tx_queue which works as acknowledgement for the host, after receiving which it(host) sends new packet. This is overall flow for storing file in the memroy.

Now to send file out on ethernet interface processor breaks file into number of packets, pops free_queue for empty buffer address and stores packet at that address. After storing packet successfully processor pushes the buffer address to tx_queue. The transmit engine running inside NIC which is polling this tx_queue reads that buffer and sends it out. One by one all the packets sent out.

4.2 Processor Accelerator Interface

The processor begins by feeding the data to the registers 1 to 12. After that, it sets the bits 0 and 2 of register 0, which signals the accelerator to start working. The processor can then continue with other tasks or poll on the register 0 as it waits for the accelerator to complete, which is indicated by setting bit 3 of register 0 as high. Then, the processor updates the registers to the parameters for the next stage or image.

In addition, the accelerator has a control daemon which resets the registers, reads the data from the AFB_ACCELERATOR_REQUEST, writes them onto the accelerator registers, and sends back the resposne to the AFB_ACCELERATOR_RESPONSE. The control daemon runs infinitely waiting on AFB_ACCELERATOR_REQUEST requests from the processor.

The accelerator also has a worker daemon, which waits for the appropriate bits to be set in the r0 register. When the processor sends the request to execute through the registers, it calls the core function with the parameters stored by the processor in other registers. When the execution is completed, it writes back 0 into bit 3 of r0, thereby signalling the

No. of active cores	Total time taken (in clock cycles)	Time per image (*10^6 clock cycles)	Utilisation (normalised with one engine)	Speedup (normalised with one engine)	Performance (GOPS (fps))
1	116,653,415	116.65	1	1	69(1.08)
2	119,693,711	59.85	0.97	1.94	134(2.09)
3	123,897,062	41.30	0.94	2.82	194(3.03)
4	133,412,216	33.35	0.87	3.48	240(3.75)
5	146,118,721	29.22	0.80	4.00	274(4.28)
6	162,244,765	27.04	0.72	4.32	296(4.63)
7	178,879,023	25.55	0.65	4.55	313(4.89)
8	201,227,739	25.15	0.58	4.64	318(4.97)

Table 4.1: Characterization of the performance of inference engine cluster

processor that the computation is completed.

The accelerator also has an interrupt daemon which waits on bit 4 of r0, and sets the signal ACCELERATOR_INTERRUPT_8 corresponding to the above bit. The interrupt signal is not currently used, but can be utilised to prevent the processor from polling on r0 while the accelerator performs the computation.

4.3 Scaling The Number Of Engines

In Chapter 3, we described how the engine can exploit data-level parallelism to obtain higher throughput. In this section, we characterize the throughput improvement obtained by exploiting the parallelism between different images by implementing multiple accelerator engines in a single cluster, each working on its own input image. The number of engines actively running were increased from one to eight, and the following results were obtained:

From the data, it is evident that we can efficiently scale the system to support 5 accelerator engines running in parallel without significant loss of performance. The performance degradation stems primarily from the use of shared ACB connection to the memory, thereby causing contention for the resource. The degradation is small for upto 5 accelerators because the memory is being utilised around 10% of the time, as seen in Table 3.1. However, beyond 5 systems, the link becomes the bottleneck leading to greater performance degradation.

The primary reason for the degradation is the use of limited memory bandwidth which is being multiplexed by all the engines, which gets intensified due to the high latency of DRAM accesses. To mitigate the issues, we have to reduce either the memory latency or

improve the memory bandwidth as seen by the engines. The first one can be obtained by using a cache, which can reduce the latency not only for kernels which are being shared by all the modules, but also the tensors whic h are accessed serially and benefit from the being loaded from the memory beforehand.

To address the more significant issue of limited memory bandwidth, we need a better memory management scheme and a network-on-chip which can provide high data rates to each of the engine when executed in the cluster. One solution is to execute the engines in a lock-step manner, where all the engines are synchronised to generate the same request simultaneously, and the request is supplied by a larger ported memory, which improves the bandwidth manifold. The disadvantage of such a technique is that it needs the engines to be synchronised, thereby needing them to be executed simultaneously.

Chapter 5

Summary

In this work, we present a hardware acceleration engine for AI/ML inference tasks. The engine is algorithmically designed using the AHIR-V2 tools, and is optimised for maximum compute-to-memory ratio using minimal on-chip buffers. We demonstrate an application of the engine by developing a image segmentation pipeline on UNET, a well-known architecture for image segmentation, on which we are able to obtain an average throughput of 85 GOPS, with the peak throughput matching the theoretical peak performance of 96 GOPS when run on a single engine synthesized at 125MHz with an average compute-to-memory ratio of 873 operations/byte (OPB). We also integrate the engine with the AJIT processor and Network Interface Controller (NIC) to generate a system on chip (SoC) capable of performing at-edge AI/ML inference tasks. We demonstrate an average performance of over 240 GOPS when employing the four engines simultaneously, resulting in an average utilisation in excess of 62% over the duration of the computation.

References

[1] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015* (N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, eds.), (Cham), pp. 234–241, Springer International Publishing, 2015.

[2] A. Kumar, "Real-world applications of convolutional neural networks," 2021.

[3] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *CoRR*, vol. abs/1511.08458, 2015.

[4] M. Hearst, S. Dumais, E. Osuna, J. Platt, and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and their Applications*, vol. 13, no. 4, pp. 18–28, 1998.

[5] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[6] D. Ulyanov, A. Vedaldi, and V. Lempitsky, "Improved texture networks: Maximizing quality and diversity in feed-forward stylization and texture synthesis," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4105–4113, 2017.

[7] X. Huang and S. Belongie, "Arbitrary style transfer in real-time with adaptive instance normalization," in *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 1510–1519, 2017.

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, (Red Hook, NY, USA), p. 1097–1105, Curran Associates Inc., 2012.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[11] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, 2016.

[12] J. Kim, J. K. Lee, and K. M. Lee, "Accurate image super-resolution using very deep convolutional networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1646–1654, 2016.

[13] Y. Choi, M. Choi, M. Kim, J.-W. Ha, S. Kim, and J. Choo, "Stargan: Unified generative adversarial networks for multi-domain image-to-image translation," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8789–8797, 2018.

[14] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. 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*, FPGA '15, (New York, NY, USA), p. 161–170, Association for Computing Machinery, 2015.

[15] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, (New York, NY, USA), p. 26–35, Association for Computing Machinery, 2016.

[16] M. Cho and Y. Kim, "Fpga-based convolutional neural network accelerator with resource-optimized approximate multiply-accumulate unit," *Electronics*, vol. 10, no. 22, 2021.

[17] S. Sahasrabuddhe, H. Raja, K. Arya, and M. Desai, "Ahir: A hardware intermediate representation for hardware generation from high-level programs," pp. 245–250, 01 2007.

[18] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, and J. Ross, "In-datacenter performance analysis of a tensor processing unit," 2017.

[19] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Automatic compilation of diverse cnns onto high-performance fpga accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 2, pp. 424–437, 2020.

[20] W. Huang, H. Wu, Q. Chen, C. Luo, S. Zeng, T. Li, and Y. Huang, "Fpga-based high-throughput cnn hardware accelerator with high computing resource utilization ratio," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 8, pp. 4069–4083, 2022.

[21] X. Wei, Y. Liang, and J. Cong, "Overcoming data transfer bottlenecks in fpga-based dnn accelerators via layer conscious memory management," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2019.

[22] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High-performance fpga-based cnn accelerator with block-floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1874–1885, 2019.

[23] S. Liu, H. Fan, X. Niu, H.-c. Ng, Y. Chu, and W. LUK, "Optimizing cnn-based segmentation with deeply customized convolutional and deconvolutional architectures on fpga," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, dec 2018.

[24] S. Liu and W. Luk, "Towards an efficient accelerator for dnn-based remote sensing image segmentation on fpgas," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 187–193, 2019.