EdgeAlign: Platform for Deep Reinforcement Learning based Genome Sequence Alignment on Edge Devices

Dual Degree Project Phase - II

Submitted in partial fulfillment of the requirements
for the degree of

Bachelor and Master of Technology
Microelectronics

by

Aryan Lall
Roll No. - 17D070053

Supervisor:
Prof. Siddharth Tallur

Department of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
Powai, Mumbai - 400076
June 2022

Dissertation Approval

The dissertation entitled **EdgeAlign: Platform for Deep Reinforcement Learning based Genome Sequence Alignment on Edge Devices** by Aryan Lall (Roll no: 17D070053) is approved for the degree of **Dual Degree (Bachelor and Master of Technology)** in **Electrical Engineering** with specialization in **Microelectronics**.

Examiners : Prof. Mukul C. Chandorkar and Prof. Rajbabu Velmurugan

Supervisor : Prof. Siddharth Tallur

Chairperson : Prof. Mukul C. Chandorkar

Date: 09-June-2022

Place: IIT Bombay

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.

Date: 09-June-2022

Aryan Lall
Roll No. 17D070053

Acknowledgements

Firstly, I thank God Almighty for giving me the strength, ability, and opportunity to undertake this research for the completion of my Dual Degree (B.Tech + M.Tech).

I would like to express my deep gratitude to Prof. Siddharth Tallur for providing me the opportunity to work on this project under his kind guidance. I have learned a lot from him and admire his integrity and dedication to research all through the project. I would like to thank Mr. Maheswar, Mr. Mahesh and several of WEL Lab staff for their kind guidance and support throughout the project work. I would like to acknowledge Prof. Kiran Kondabagil and Haystack Analytics for their help and support related to the Genomics application. Special thanks to Mr. Parimal and Mr. Aniket from the Xilinx family for their valuable inputs on the Hardware design.

Aryan Lall

Abstract

This report presents our work towards the development and demonstration of the toolchain for deploying the machine learning frameworks on edge devices, including the resource-constrained FPGAs. This includes the comparative analysis of the performance obtained on different hardware platforms using benchmark applications. We have also demonstrated a Deep Reinforcement Learning-based Genomics application of Pairwise DNA sequence alignment on the edge devices by utilizing the above toolchain. Lastly, we have also worked towards the design and implementation of the CNN hardware accelerators for improving our application inference timings, using which we have successfully achieved an acceleration of about 2X regardless of various bottlenecks as discussed later in our work.

The first section of this report provides an overview of TinyML, its applications, and the TensorFlow Lite framework. Further, the report addresses the development of the TinyML toolchain, including the implementation details and results obtained on different hardware platforms including ST microcontrollers and FPGAs. In the later section, we lay out our work towards the implementation of the DNA pairwise sequence alignment application and its deployment on edge devices. Finally, we conclude our work by highlighting various designs of the CNN hardware accelerators and their integration with our genomics application.

Our work would prove crucial for implementing various machine learning-based applications on edge devices and optimizing the obtained performance. This makes machine learning even more accessible and affordable than before. At the end, our future is contained in smaller and intelligent devices! which would ultimately lead to a true human-machine friendship. All the required steps have been properly documented to enable quicker reproduction of the aforementioned work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 TinyML and its applications

Machine learning has transformed the way we visualize the data and offers promising solutions to a vast variety of problems. These data-driven approaches demand large computational power and storage for their handling and analysis. This poses a problem for their implementation on low-power edge devices which are powered by Microcontroller Units (MCUs).

The TinyML paradigm proposes to integrate Machine Learning(ML)-based mechanisms within small objects powered by MCUs [1]. This provides the door for the creation of new apps and services that do not require cloud processing assistance which on the other hand consumes higher energy, contributes to latency, and poses data security and privacy issues. The incorporation of intelligence into these tiny devices has a number of benefits which include energy-efficient devices, cheaper and faster product which offers data security. However, considering the enormous heterogeneous range of existing devices with limited computational power and memory, it still remains a challenge to implement machine learning frameworks on these tiny devices. With great power, comes great responsibilities!

The applications of TinyML can range from smart devices to the recent booming industry of AgroTech. Many IoT applications are already integrated in our daily personal and professional activities. Health sector offers large opportunities where applications such as health monitoring, visual assistance, hearing aids, personal sensing, activity recognition, etc. requires reliable smart devices. Apart from these, few other applications from the never-ending list includes augmented reality, real-time voice recognition, language translation, context-aware support systems, machine

health monitoring, Hearables [2], etc. Such low-power requirement even allows the operation of these devices in remote areas. Recently, machine learning is also paving its way in bioinformatics and a lot of opportunities exist in the field of genomics. Later in this report, we will discuss the implementation of a genomics-based application of **DNA pairwise sequence alignment** on the edge devices. TinyML would be the next wave of digital revolution.

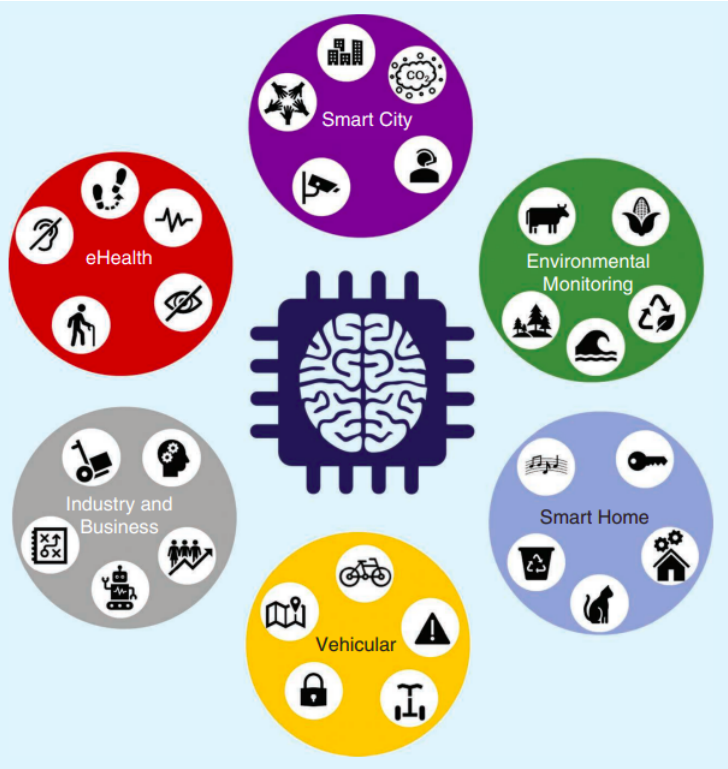


Figure 1.1: Applications of TinyML [Source [1]]

1.2 TensorFlow Lite for Microcontrollers

1.2.1 Overview

TensorFlow Lite Micro (TFLM) is an open-source ML inference framework for running deep-learning models on embedded systems [3]. This framework employs an unified flexible interpreter which addresses the resource-constraints and heterogeneity of the embedded platforms. Beyond deploying a model to an embedded target, the framework must also have a means of training a model on a higher compute plat-

form. The training can thus be performed using the tensorflow APIs with maximum efficiency, and TFLite micro can further be used as the inference engine in MCUs. It doesn't require operating system support, any standard C or C++ libraries, or dynamic memory allocation and also supports floating and quantized (int8, uint8) inference.

TFLM makes it easy to get TinyML applications running across architectures, and it allows hardware vendors to incrementally optimize kernels for their devices. It gives vendors a neutral platform to prove their performance and offers countless benefits which include portability, flexibility, minimization of external dependencies, custom accelerators, etc.

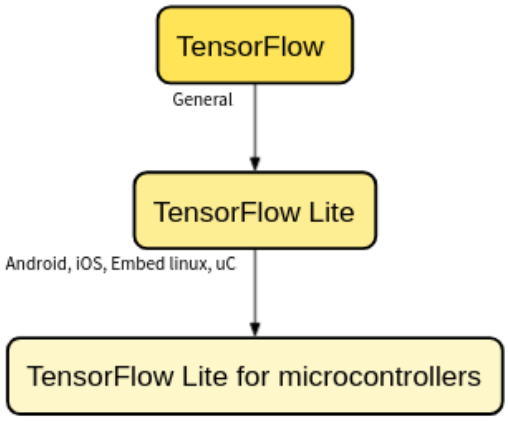


Figure 1.2: TensorFlow Lite framework for microcontrollers can be thought as a subset which utilizes the features offered by the higher TensorFlow APIs in a limited fashion. TensorFlow Lite supports a limited subset of TensorFlow operations, with limited set of devices.

TensorFlow Lite for Microcontrollers is written in **C++11** and requires a **32-bit** platform. It has undergone testing and it has been deployed extensively with many processors based on the Arm Cortex-M architecture. It has been ported to other architectures including ESP32 and many digital signal processors (DSPs). The framework is also available as an Arduino library. It can generate projects for environments such as Mbed as well.

1.2.2 Implementation Workflow

Once the model training has been completed using the tensorflow framework or the high-level APIs such as Keras, a tensorflow model is obtained. To convert this

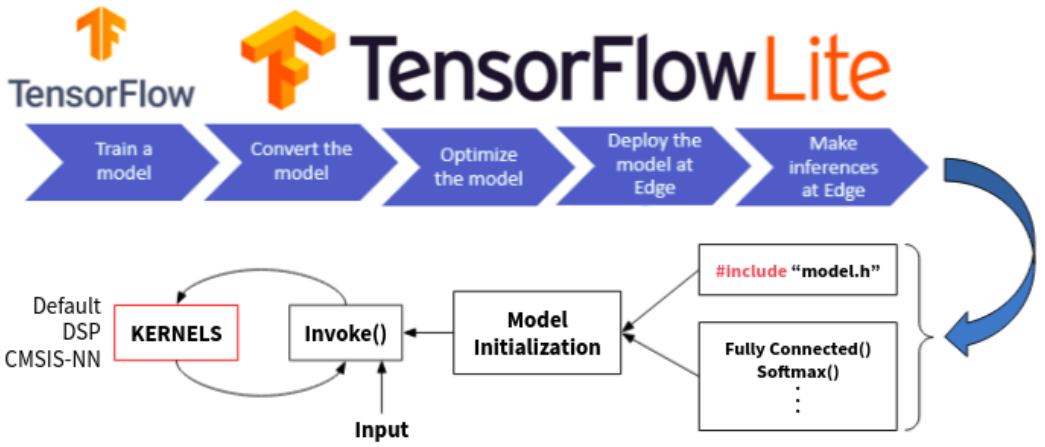


Figure 1.3: Implementation workflow for TensorFlow Lite for Microcontrollers

trained model to run on microcontrollers, it needs to be converted into TensorFlow Lite model. On Python, this conversion can be carried out using the TensorFlow Lite converter Python API [4]. This will convert the model into a FlatBuffer, reducing the model size, and modify it to use TensorFlow Lite operations.

Further, the users can carry out Post-training quantization which can reduce the model size while also improving the CPU and hardware accelerator latency. All the model parameters can be expressed using 8-bit integral or half-precision floating representations. However, this can have trade-off and might reduce the model accuracy, although this downgrading effect is negligible and can be tolerated. This technique can drastically reduce the latency while implementing applications on microcontrollers or FPGAs as integer arithmetic operations are much faster as compared to the corresponding floating operations. This quantization can be carried out in python itself while converting the tensorflow model to TFLite model [5]. Further, this TFLite model is converted to a C byte array (flatbuffer) using standard tools to store it in a read-only program memory on device. Figure 3.1 shows the workflow for the TFLite model deployment. *model.h* file contains the final C byte array which represents the entire model. Further, the individual model layers or operations can be registered and the model can then be initialized using the registered operations and the flatbuffer. Finally, we can invoke the inference engine using the user-inputs.

Each layer operation (fully connected, etc.) requires a Kernel for execution. These individual kernels are by-default written in C++11, and thus they can be accelerated on hardware using DSP instructions or libraries such as CMSIS-NN (for Cortex-M processors), etc. More details would be discussed in further sections.

1.2.3 Obtaining the TFLite model file

Once we have trained our tensorflow model on a host PC, we need to convert this model into a special, space-efficient format for use on memory-constrained devices. We'll use the TensorFlow Lite Converter for this purpose. For a simple demonstration, the following python code coverts a trained keras model to a TFLite model.

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
tflite_model = converter.convert()
open("my_model.tflite", 'wb').write(tflite_model)
```

As we have discussed in the previous section, we can obtain an 8-bit integer(int8) quantized model by making the following changes in the converter configuration.

```
# Provide a representative dataset to ensure correct quantization of input values
def representative_data():
  for input_value in tf.data.Dataset.from_tensor_slices(training_data).batch(1).take(100):
    yield [input_value]

converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data
# Ensure that if any ops can't be quantized, the converter throws an error
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
# Set the input and output tensors to int8
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8

tflite_model_quant = converter.convert()
open("my_model_quant.tflite", 'wb').write(tflite_model_quant)
```

Finally, we need to convert the TensorFlow Lite model (.tflite) into a .h header file that can be loaded by TensorFlow Lite for Microcontrollers. The *xxd* utility in linux can very conveniently perform this operation as follows-

```
# Install xxd if it is not available
$ apt-get update && apt-get -qq install xxd
# Convert to a C source file (TFLite micro model)
$ xxd -i 'my_model.tflite' > 'model.h'
```

This *model.h* file contains a C byte array of our neural network model in the FlatBuffer format. This would be further used while configuring the microcontroller application project.

Chapter 2

Literature Review

Since we would be demonstrating our toolchain using the application of DNA pairwise sequence alignment, we conducted a literature survey to identify the existing implementations and their salient features. Most of them are based on improving the traditional methods, and a few of them uses the technique of deep learning. The sequence alignment is an NP-complete problem. Therefore, many heuristic approaches are proposed for solving this particular important bioinformatic problem.

Mircea et al. [16] presented one of the first RL implementations of the DNA sequence alignment. They used a tabular approach to the problem and offered a first Markov decision process (MDP) to solve it. Their method yielded promising results on some test data, and the findings were marginally enhanced in a second paper [17] using a slightly adjusted scoring algorithm. Jafari et al. [19] introduced DQNs and an actor-critic algorithm in their work. Their DQN model architecture was based on Long short-term memory (LSTM) network which involves sequential computation. However, both did not publish their code or implementation details to reproduce the results and test it on larger benchmark datasets.

Song et al. [18] provided a comprehensive implementation of the pairwise sequence alignment. They also used various pre-processing techniques such as Clustal, MUMmer, etc. to improve the alignment results, and highlighted the influence of various system parameters such as the learning rate, window sizes, etc. over the accuracy. They also proposed a CNN based Dueling Double Deep Q-network for their RL model architecture. Ramakrishnan et al. [20] applied deep reinforcement learning to the model of the Phylo game for multiple sequence alignment (MSA). They used a convolutional neural network (CNN) based architecture for their actor-critic model. However, this model is too detailed to be applied to real-world problems with dozens of sequences and hundreds of molecules. Joeres et al. [21] offered

a comprehensive analysis of the performance of different RL algorithms in MSAs. They also provided insight over why RL algorithms can be slower as compared to the traditional implementations. All of these implementations are based on the technique of deep reinforcement learning, and none of them are deployed on the edge devices.

Several non-machine learning based implementations had also been proposed for improving the sequence alignment method including few multi-threads based implementations on GPUs [22, 23, 24] for improving the throughput. ClustalW [25] is a well-known tool for solving the MSA problems. It also has a quick algorithm and also reasonable processing time for large datasets. ClustalW uses progressive alignment method. It determines the best alignment by matching the sequences that are the most similar first, then moving on to the sequences that are the least similar.

In particular, several pairwise alignment algorithms, such as banded alignment, the BLAST, and the MUMmer have been proposed to improve the speed of pairwise alignment [26, 27, 28]. These alignment approaches aimed to address the issue of complexity by restricting the alignment's range and then extending it after word matching or average common substring matching. However, there were some accuracy issues when expanding small local alignments to large and complex sequences in these circumstances.

Regarding the deployment of TinyML applications on the MicroBlaze, we weren't able to find any published references. The CNN hardware accelerator design in our work is influenced from the Eyeriss [41] architecture, which is an energy-efficient reconfigurable accelerator for deep convolutional neural networks. We didn't find any references which discusses about the HLS implementation of such a design.

Chapter 3

Implementation on STM32 boards

The STM32 eval boards have been designed as a complete demonstration and development platform for the STM32 MCUs and MPUs. STM32 is a family of 32-bit microcontroller integrated circuits by STMicroelectronics. These chips are grouped into related series that are based around the same 32-bit ARM processor core, such as the Cortex-M33F, Cortex-M7F, Cortex-M4F, Cortex-M3, Cortex-M0+, or Cortex-M0. Internally, each microcontroller consists of the processor core, static RAM, flash memory, debugging interface, and various peripherals [6]. This offers very high performance, real-time capabilities, digital signal processing, low-power operation, and connectivity, while maintaining full integration and ease of development.

We have started out by implementing the TFLite framework on the STM32 boards. Following are the boards on which we tested our implementation -

MCU	**Core**	**RAM(kB)**	**Flash(kB)**	**DSP & FPU**	f_{MAX}**(MHz)**
STM32F401RE	Cortex-M4	96	512	Yes	84
STM32F746NG	Cortex-M7	320	1024	Yes	216

Table 3.1: Specifications for the tested STM32 boards

The above boards were readily available at the time of project commencement and they also support the DSP instructions, thus allowing custom acceleration of the execution kernels. Both of these boards can be easily programmed/debugged using the mini-USB cable and the applications were developed using the STM32 Cube IDE. This eclipse based environment simplifies the project creation and development. The first step towards the implementation of TFLite framework involves the generation of source code files which can be used as a library. Further details are provided in the following sections.

3.1 Generating source files for TFLite

TensorFlow Lite for Microcontrollers is able to generate standalone projects that contain all of the necessary source files, using a Makefile. The current supported environments are Keil, Make, and Mbed. By default, the project will be compiled for the host operating system, although we can specify a different target architecture. The original guide for the TensorFlow lite micro uses the Make build tool to generate a number of example projects that we can use as templates for our microcontroller projects. The idea is to use the TensorFlow Lite as a library instead of a starter project.

For a successful execution of Make, we require few additional tools which can be installed via the following set of commands.

```
$ sudo apt update
$ sudo apt install make git python3.7 python3-pip zip
```

Further we can clone the newest version of the TensorFlow repository.

```
$ git clone --depth 1 https://github.com/tensorflow/tflite-micro.git
```

The last step would be to navigate to the cloned directory and run the Makefile in the TensorFlow Lite for Microcontrollers directory.

```
$ cd tflite-micro
$ make -f tensorflow/lite/micro/tools/make/Makefile
    OPTIMIZED_KERNEL_DIR=cmsis_nn generate_projects
```

Once this is completed, it generates a number of example projects at *tensorflow/lite/micro/tools/make/gen/linux_x86_64/prj*. The exact path would depend on the host operating system. Since, we were using Linux to build the applications, we have *linux_x86_64/* in our directory path. Inside the desired project e.g *hello_world*, we have projects created for the current supported environments such as Keil, Make, and Mbed. Since Make projects are target neutral, we would navigate to this directory. This contains the two most important folders (*tensorflow* [1] and *third_party* [2]) which consists of all the required TFLite source and header files, and these would be further imported in the application project as libraries.

An important flag to notice while running the Make command is the OPTIMIZED_KERNEL_DIR option. Using this option generates the required CMSIS_NN kernel files and they are located in the *tensorflow/lite/micro/kernels* directory of the above generated tensorflow[1] folder. The value of this flag (*cmsis_nn*) is the folder name where the CMSIS_NN kernels are stored. TFLite source files are now generated :)

3.2 Configuring Application project

We can create an application project in the STM32 Cube IDE with C++ as the target language. Make sure to enable required peripherals/pins and configure them properly in the software application code. For example, we require a Timer module for profiling the run-time execution. We can enable any of the available timer modules and instantiate it with proper configuration values such as prescalar, period, etc.

Once the project has been created, we can configure it for accommodating and compiling our tensorflow model using the following steps -

Copying the TensorFlow Lite model & source code files to our project:
We need to copy our TFLite model file (*model.h*) into the include directory of the project. For STM32 Cube IDE project, this is located at *<project_directory>/Core/Inc*.

Further, we create a new folder *tensorflow_lite* in the *<project_directory>* and copy the two important folders that we created earlier (*tensorflow* [1] and *third_party* [2]) from the *hello_world* project at this location. We must also delete the examples folder located in the *<project_directory>/tensorflow_lite/tensorflow/lite/micro* as it contains unnecessary C files which must be avoided while compiling. The project directory structure should finally appear as follows -

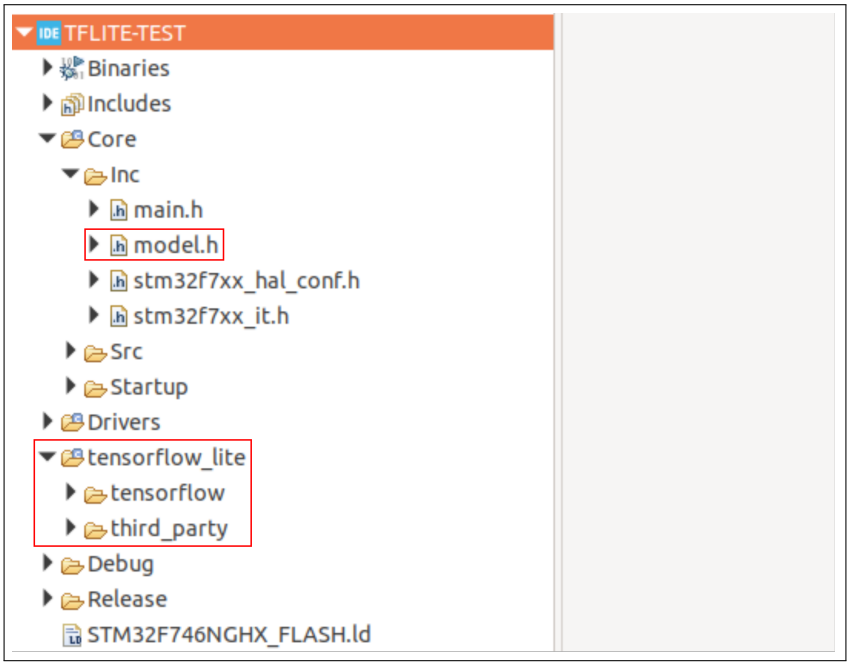


Figure 3.1: Project directory structure

Now, we would only make required changes in the created *tensorflow_lite* folder.

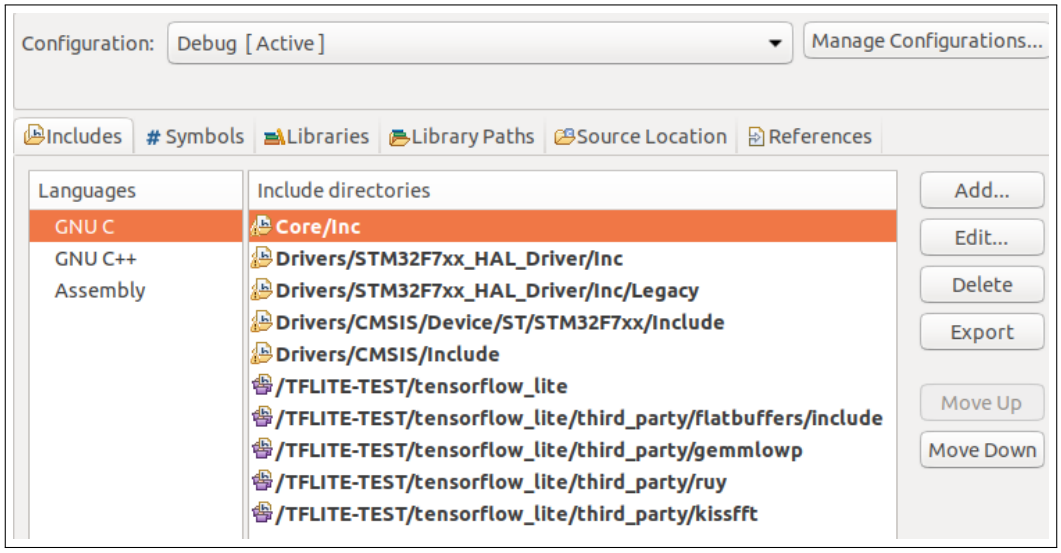
Include Headers and Source files in build process:

Even though the source files are located in our project, we still need to tell our IDE to include them in the build process. Go to **Project >Properties**. In that window, go to **C/C++ General >Paths and Symbols >Includes tab >GNU C**. Click **Add** and in the pop-up window, click **Workspace**. Select the ***tensorflow_lite*** directory in the project. Check **Add to all configurations** and **Add to all languages**.

Similarly, add the following directories also -

- *tensorflow_lite/third_party/flatbuffers/include*
- *tensorflow_lite/third_party/gemmlowp*
- *tensorflow_lite/third_party/kissfft*
- *tensorflow_lite/third_party/ruy*

Make sure that these include directories are also reflected in the **GNU C++** and **Assembly** languages, for each of the **Debug** and **Release** configuration. Finally, we would include the source directory of TFLite. Go to the **Source Location** tab and add *<project_directory>/tensorflow_lite* to both the Debug and Release configurations. Click Apply and Close.



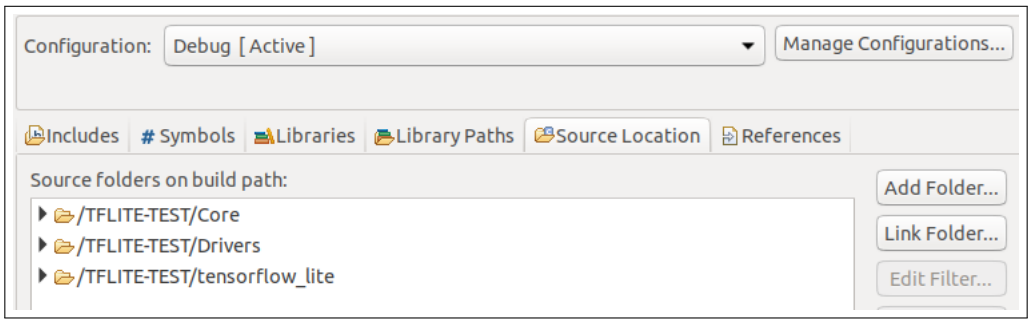


Figure 3.2: Includes and Source Location tabs after above changes

The required header and source files are now included in our project. Thus, we can easily use the TFLite framework as a target neutral library for various kind of projects. Lastly, we need to delete few unnecessary files before we compile the code. Following folders are required to be deleted (*these files will be used later) -

1. *<project_directory>/tensorflow_lite/tensorflow/lite/micro/tools/make/downloads/cmsis/CMSIS/NN*

2. *<project_directory>/tensorflow_lite/tensorflow/lite/micro/kernels/<cmsis_nn folder >*

Writing the Application code:

We can now edit the *main.cpp* file and add our application code. For using the tensorflow variables and functions, we need to include the following header files -

```
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"

#include "model.h"  // Include the model file
```

Depending on the hardware platform, we need to initialize the peripherals with proper configuration values. The STM Cube IDE automatically generates these initialization functions and thus, they can be easily called inside the code. Rest of the discussion focuses on explaining the different sections inside the TFLite application code. For an example code, please refer to main.cpp.

First, we initialize the TFLite micro model with the parameters of our model file.

```
model = tflite::GetModel(my_model);  //my_model is the C byte array
```

Next, we must register the layer operations that our model uses. This will be used by the interpreter to access the operations that are used by the model. We can do this in two different ways:
1. Include all the operations available in tensorFlow lite for microcontrollers. Although this is an easier way to load the operations, it uses a lot of memory as various unnecessary operations are also included. This method is not recommended for microcontroller applications. Following code shows how to declare the operation resolver using this method -

```
tflite::AllOpsResolver resolver  // All operations registered;
```

2. We can manually register only those operations which are required by the model. Since a given model will only use a subset of these operations, it's recommended that real world applications load only the operations that are needed. This is done using a different class, *MicroMutableOpResolver* as follows -

```
static tflite::MicroMutableOpResolver<1> micro_op_resolver;
tflite_status = micro_op_resolver.AddFullyConnected();
if (tflite_status != kTfLiteOk)
{
  error_reporter->Report("Could not add FULLY CONNECTED op");
}
```

Next, we would build the interpreter to run the model and allocate memory for the required tensors as follows -

```
static tflite::MicroInterpreter static_interpreter(
    model, micro_op_resolver, tensor_arena, kTensorArenaSize,
       error_reporter);
interpreter = &static_interpreter;

tflite_status = interpreter->AllocateTensors();
```

Finally, we can assign the user values to the input tensor and invoke the TFLite model to obtain the output results.

```
model_input->data.f[0] = 2.0f;  // User value
tflite_status = interpreter->Invoke();   // Invoke TFLite model
float my_output = model_output->data.f[0];  // Get output
```

Depending on the application, the above code can be modified for implementing complex algorithms. For example. we have also implemented the MNIST classification problem on the STM Disco board.

Miscellaneous:

We can also modify the Debug code of tensorflow lite to allow error messages to be sent over the UART peripheral, thus making the debugging easier. Open the file *<project_directory>/tensorflow_lite/tensorflow/lite/micro/debug_log.cc* and update it's code as follows -

```
#include "tensorflow/lite/micro/debug_log.h"
extern "C" void __attribute__ ((weak)) DebugLog(const char* s){
  // To be defined by the user further
}
```

Go to the *main.cpp* file and redefine this debug function as follows -

```
extern "C" void DebugLog(const char* s)
{
 // Enable error messages to be sent via UART
 HAL_UART_Transmit(&huart1, (uint8_t *)s, strlen(s), 100);
}
```

In STM32 Cube IDE, *printf* (and variants) does not support floating point values. To add that, we need to go to *Project >Properties >C/C++ Build >Settings >Tool Settings tab >MCU G++ Linker >Miscellaneous*. In the *Other flags* pane, add the line **-u_printf_float** for both Debug and Release configurations. Finally, we can build our project and generate the .elf file.

Running the application:

Once the application has been built successfully, we can run it on the STM32 board. We can use utilities such as PuTTY to send or receive the UART messages over the serial terminal. In python, pySerial can be used for communicating with the board over the serial port.

```
arm-none-eabi-objdump -h -S  TFLITE-TEST.elf  > "TFLITE-TEST.list"
arm-none-eabi-objcopy  -O binary  TFLITE-TEST.elf  "TFLITE-TEST.bin"
   text    data     bss     dec     hex filename
  95964  205748   53464  355176   56b68 TFLITE-TEST.elf
```

Figure 3.3: Size (bytes) of different sections in the .elf file (dec is total size)

3.3 Acceleration using DSP instructions

Since the TFLite kernels are written in C++, it gives us an opportunity to modify and accelerate them using various hardware optimizations. This enables us to completely utilize the power of the processor core and leverage the performance of our application. One such optimization could be to use the available DSP instructions which many of the ARM cores offer.
The **CMSIS DSP** software library is a suite of common signal processing functions for use on Cortex-M and Cortex-A processor based devices. This library has generally separate functions for operating on 8-bit integers, 16-bit integers, 32-bit integer and 32-bit floating-point values. The performance gain would vary across different processor architecture. Rest of the section discusses about the usage of DSP instructions and how can we modify the TFLite kernels. This discussion **assumes** the following -

1. Cortex-M or Cortex-A processor based device which supports DSP instructions
2. User has configured the application project according to section 3.2

3.3.1 Usage in Fully Connected layer

A fully connected layer essentially involves the following 3 stages of computation -

1. A matrix multiplication (weight x input)
2. Addition of 2 vectors (bias and output of previous stage)
3. Apply layer activation function (ReLU, Softmax, etc.)

The first 2 stages of the fully connected layer can be effectively implemented and accelerated using the common CMSIS DSP instructions. The implementation of the last stage using DSP is subject to complexity and availability of the DSP instructions.

Considering that we are working with floating-precision, the following CMSIS DSP functions can help us to accelerate the computation of the fully connected layer -

1. **arm_mat_mult_f32** : This function performs a floating-point matrix multiplication using DSP instructions.
2. **arm_add_f32** : This function performs a floating-point addition between 2 input vectors using DSP instructions.

The matrix input is provided in a special format of matrix data structures, called as an ARM matrix instance. Thus, this matrix instance is first initialized using a function (*arm_mat_init_f32)* which sets the values of the internal structure fields.

3.3.2 Configuring project settings

Before we modify and compile the code, we are required to make few changes in the project settings such as including DSP functions header files, setting preprocessor symbols, etc. These are listed as follows -

Including the DSP Header files:
There are two ways to obtain the CMSIS DSP header files. First, the TFLite make process itself generates the DSP header files and they are located at *<project_directory>/tensorflow_lite/tensorflow/lite/micro/tools/make/downloads/cmsis/CMSIS/DSP/Include*.

Second, we can clone the original github repository of CMSIS. Further, we need to copy the folder *CMSIS_5/CMSIS/DSP/Include* and paste it under *<project_directory>/Drivers/CMSIS/DSP* (new folder).

Go to **Project >Properties**. In that window, go to **C/C++ General >Paths and Symbols >Includes tab >GNU C**. Click **Add** and in the pop-up window, click **Workspace** and select **only** one of the above folders in the project. Check **Add to all configurations** and **Add to all languages**.

Including the DSP Libraries:
We need to include and link the appropriate library in the application. Copy the folder *<STM32Cube_Repository >STM32Cube_FW_F(2/4/7)_V.X.XX.X/Drivers/CMSIS/Lib* and paste it under *<project_directory>/Drivers/CMSIS*. The exact path (2/4/7) would depend on whether Cortex-M(3/4/7) is used, respectively.
In the properties window, go to **C/C++ Build >Settings >MCU G++ Linker > Libraries**. In the Library search path section, click Add and select the GCC library present in the workspace path *<project_directory>/Drivers/CMSIS/Lib/GCC*. Now, to add the specific library to work with, click Add in the Libraries section and insert **arm_cortexM7lfsp_math** (for Cortex-M7) or **arm_cortexM4lf_math** (for Cortex-M4). For more details on which library to include, refer to comments in *arm_math.h* file.

Insert Preprocessor symbols:
Finally, we need to inform the compiler about the presence of DSP math instructions by inserting appropriate preprocessor symbols. Go to **C/C++ Build >Settings >MCU GCC Compiler >Preprocessor** and add **ARM_MATH_CM(3/4/7)** to the Define symbols section. The exact symbol (3/4/7) would depend on whether Cortex-M(3/4/7) is used, respectively. Click Apply and Close.

3.3.3 Modifying the TFLite Kernel

The kernel (*FullyConnected* function) which performs the floating-precision fully connected layer operation is located at -
<project_directory>/tensorflow_lite/tensorflow/lite/kernels/internal/reference/fully_connected.h

By default, the kernel code uses normal *for* loops for computation as follows -

```
for (int b = 0; b < batches; ++b) {
  for (int out_c = 0; out_c < output_depth; ++out_c) {
    float total = 0.f;
    for (int d = 0; d < accum_depth; ++d) {
      total += input_data[b * accum_depth + d] *
               weights_data[out_c * accum_depth + d];
    }
    float bias_value = 0.0f;
    if (bias_data) {
      bias_value = bias_data[out_c];
    }
    output_data[out_c + output_depth * b] =
       ActivationFunctionWithMinMax(
        total + bias_value, output_activation_min,
           output_activation_max);
  }
}
```

Figure 3.4: Original kernel code

The value of variable *batches* is generally 1. Thus, we can ignore this variable for simplifying the loop. The *input_data* vector has *accum_depth* number of elements. Similarly, both the *output_data* and *bias_data* vector has *output_depth* number of ele-

ments. This example uses the ReLU activation. Instead of iterating over each element and performing the computation, we can replace the above code section with few DSP instructions as follows -

```
arm_matrix_instance_f32 Weight, Input, Output;

// Initailize matrix instance
arm_mat_init_f32(&Weight, output_depth, accum_depth, (float32_t*)
    weights_data);
arm_mat_init_f32(&Input, accum_depth, 1, (float32_t*)input_data);
arm_mat_init_f32(&Output,output_depth,1, (float32_t*)output_data)

// Matrix multiplication
arm_mat_mult_f32(&Weight, &Input, &Output);

// Addition of bias
if (bias_data) {
  arm_add_f32(output_data, const_cast<float*>(bias_data),
      output_data, output_depth);
}

// ReLU Activation
for (int out_c = 0; out_c < output_depth; ++out_c) {
  output_data[out_c] = ActivationFunctionWithMinMax(
                output_data[out_c], output_activation_min,
                    output_activation_max);
}
```

Figure 3.5: Updated kernel code using DSP instructions

For more information regarding the DSP function arguments and details, please refer to the CMSIS DSP manual. The above example considers a floating operation, however, we can also work with fixed-precision. The DSP library has separate functions for operating on 8-bit integers, 16-bit integers, 32-bit integer and 32-bit floating-point values. Thus, we can modify the kernel accordingly and accelerate the layer operations.

3.4 Acceleration using CMSIS-NN Library

Another optimization technique for accelerating the TFLIte kernels is to use the CMSIS NN library for Cortex-M processor cores, which is a collection of efficient neural network kernels developed to maximize the performance and minimize the memory footprint of neural networks. This library has separate functions for operating on different weight and activation data types including 8-bit integers (q7_t) and 16-bit integers (q15_t). The description of the kernels are included in the function description. For more implementation details, refer to the paper [7].

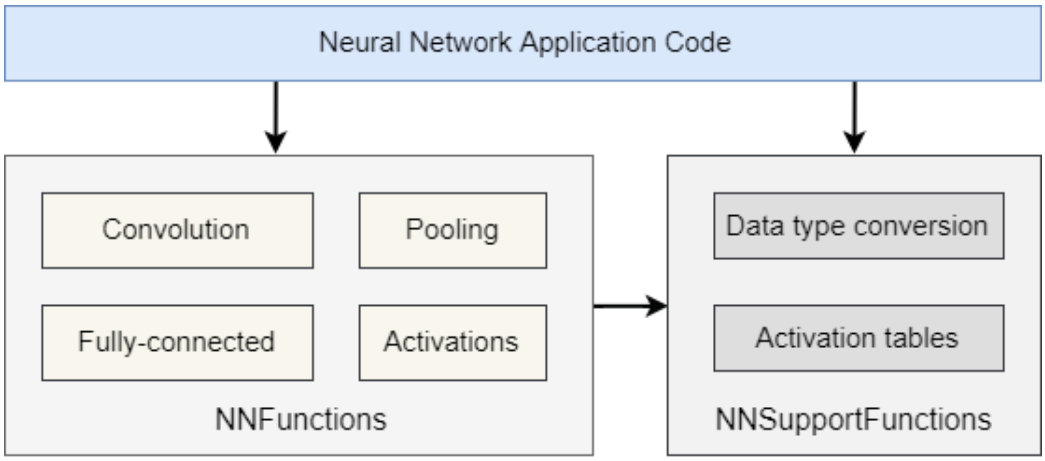


Figure 3.6: CMSIS NN Block diagram. Invoking the model executes the pre-build NN functions.

Various experimental results [7] have shown that neural network inference based on CMSIS-NN kernels achieves **4.6X** improvement in runtime/throughput and **4.9X** improvement in energy efficiency. We can directly use this kernels in our application code to implement neural network algorithms on Arm Cortex-M CPUs. However, the current library doesn't offers support for architectures such as RNNs, Transformers, etc. The performance gain would again vary across different processor architecture. Rest of the section discusses about the usage of these kernels. This discussion **assumes** the following -

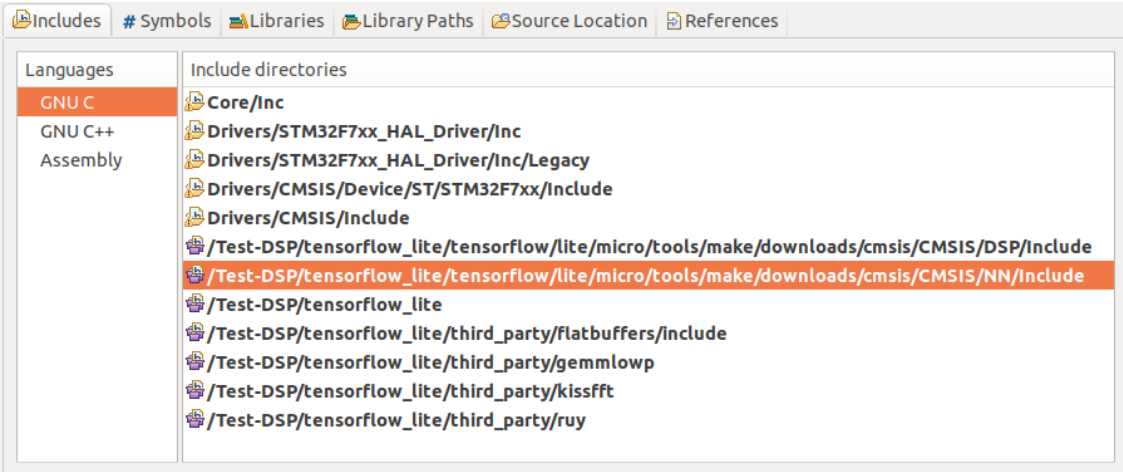
1. Cortex-M processor based device which supports DSP instructions
2. User has configured the application project according to section 3.3.2

3.4.1 Configuring project settings

Restore the folders that were deleted while configuring the project in this section. Both of these folders contain the CMSIS NN kernel source files and thus are required to be added in the project. Configure the project as follows -

Including the Header files:

All the header files are present in the above ***NN/Include*** folder which are auto-generated during the make process of TFLite. However, we can also obtain the header and source files from the CMSIS github repository, as discussed in the DSP section. Include this folder in the project. Check **Add to all configurations** and **Add to all languages**.



Insert Preprocessor symbols:

We need to inform the compiler about the presence of CMSIS_NN kernels by inserting appropriate preprocessor symbols. Insert the following symbols in the project, similar to the way discussed in the DSP section. Click Apply and Close.

1. **CMSIS_NN**
2. **ARM_MATH_DSP** // DSP instructions are supported
3. **__FPU_PRESENT=1** // FPU is present in the device core

For more information on which symbols to include, check out the comments in the file *arm_nnfunctions.h*.

3.4.2 Modifying the TFLite Kernel

The CMSIS NN kenels for TFLite are auto-generated during the make process and they are located at *<project_directory>/tensorflow_lite/tensorflow/lite/micro/kernels/<cmsis_nn folder >*. Consider that we are modifying the kernel for the fully connected layer. The default kernel is located at *<project_directory>/tensorflow_lite/tensorflow/lite/micro/kernels/fully_connected.cc* and this needs to be **replaced** with the one located in the *cmsis_nn* folder. Similarly, we can replace other default kernels with the CMSIS NN kernels. Note that few of the header includes might require some modifications. For example, originally the following include line is present in the file *fully_connected.cc* -

```
#include "CMSIS/NN/Include/arm_nnfunctions.h"
```

This needs to be changed as follows -

```
#include "arm_nnfunctions.h"
```

Once all the required kernels are replaced, we can compile our application project and generate the .elf file. The performance comparison between the different optimization techniques and the resultant performance gain is discussed in the later chapters. This chapter explains the way users can configure their project settings to accommodate the TFLite framework and introduces various techniques for run-time acceleration on the STM32 boards. Users are encouraged to explore new techniques and ways to implement their own kernels and integrate them with the existing setup.

Chapter 4

Implementation on ARM Soft CPU

The last chapter was centered around the implementation of TFLite framework on hard processor cores such as ARM Cortex-M based devices, etc. Thus, we were only required to develop the software and deploy it on the MCU. However in this case, we are highly restricted by the features offered by the MCU vendor and it becomes very difficult to perform any changes on the available hardware.

FPGAs, on the other hand, offer enough flexibility to define and reconfigure the hardware. The parallel processing ability offered by FPGAs allows the development of custom hardware accelerators and is ideal for applications including image processing or artificial intelligence. Complex tasks are often solved by software implementations with fast processors. FPGAs offer a cost-effective alternative, which, via parallelization and adaption to the application, provide a significant speed advantage compared to processor-based solutions. At this point, we may ask a question, Can the machine learning frameworks be implemented on FPGAs?

Few SoC and MPSoC families from Xilinx such as Zynq, UltraScale, etc. combines the power of FPGA with the ARM core processor along with various other peripherals. This offers more design choices to the engineers and this single device forms an even more powerful embedded computing platform. For these powerful boards, we already have development environments such as **Vitis AI** which is a Xilinx's development platform for AI inference on Xilinx hardware platforms, including both edge devices and Alveo™ cards. It consists of optimized IP, tools, libraries, models and is designed with high efficiency and ease-of-use. However, not every FPGA would come with an ARM hardcore processor, especially if it is resource-constrained like CMOD A7-35T board, etc. Thus, the challenge in such devices is the absence of a hardcore PS which is required to perform complex control tasks and prepare the ground for the TFLite runtime.

4.1 ARM Cortex-M on FPGA

The ARM DesignStart FPGA program offers instant and free-of-cost access to Arm Cortex-M soft CPU IP for FPGA designs. There is no license fee and zero royalty associated with DesignStart FPGA so the developers can get started with prototyping and designing commercial products instantly [8]. **Cortex-M3** and **Cortex-M1** soft CPU IPs are available for integration in the FPGA design. Rest of the discussion is based on Cortex-M3 IP, although the project can be modified similarly for using the Cortex-M1 variant.

The Cortex-M3 is a low-power processor that features low gate count, low interrupt latency, and low-cost debug. It is intended for deeply embedded applications that require optimal interrupt response features. The processor implements the ARMv7-M Thumb instruction set, and is binary compatible with the instruction sets and features implemented in other Cortex-M profile processors. On the expense of flexibility offered by these IPs, they also have few **limitations**. The maximum supported clock frequency for Cortex-M3 IP is **50 MHz** and for Cortex-M1, it is 100 MHz. Thus, the slower clock would result in higher latency as compared to the hard-core solutions. Also, these soft CPUs don't have a floating point unit (FPU) and thus, usage of libraries such as CMSIS NN, DSP functions are not allowed. Currently the flow to update a bitstream with new Instruction Tightly Coupled Memory (ITCM) data only supports memory sizes in the range **16KB** to **128KB**. For more information regarding the soft IP, visit the ARM designstart FPGA website.

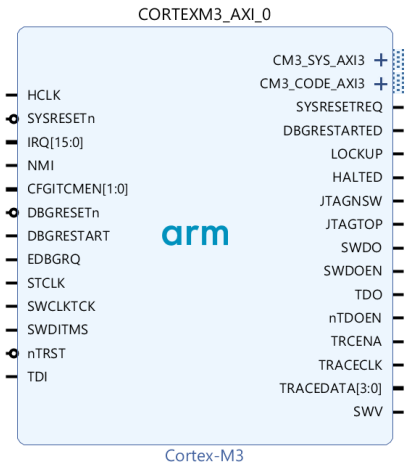


Figure 4.1: Cortex-M3 soft IP

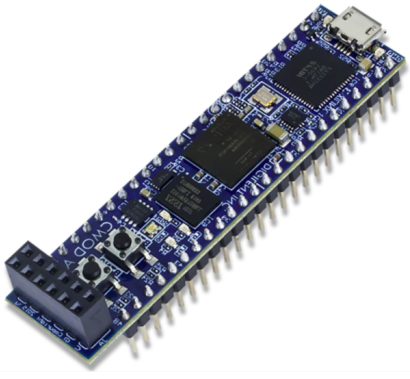


Figure 4.2: CMOD A7-35T Board

The idea is to implement the Cortex-M3 soft CPU IP on a resource-constrained

FPGA board such as Digilent's **CMOD A7-35T**, and further implement the TFLite framework for running various ML applications. The advantage of doing such a thing is that even these smaller boards would also be introduced to the machine learning domain, which wasn't possible earlier. Few specifications of the CMOD A7-35T board are mentioned below -

LUT	Flip-Flop	Block RAM	SRAM	Quad-SPI Flash
20800	41800	225 KB	512 KB	4 MB

Table 4.1: Key specifications of CMOD A7-35T board

4.1.1 Implementation Workflow

The first step towards the implementation of Cortex-M soft IP on FPGA involves the configuration of hardware and the generation of bitstream. We would be using Xilinx **Vivado 2019.1** for this purpose. The Cortex target's program memory is local to the FPGA, and the program memory contents is included in the FPGA bitfile. Xilinx uses an **MMI** file to describe the location of internal FPGA memories, so that their contents may be updated. Thus, we would also generate a .mmi file which would be later used to directly update the bitstream without the need of regenerating it.

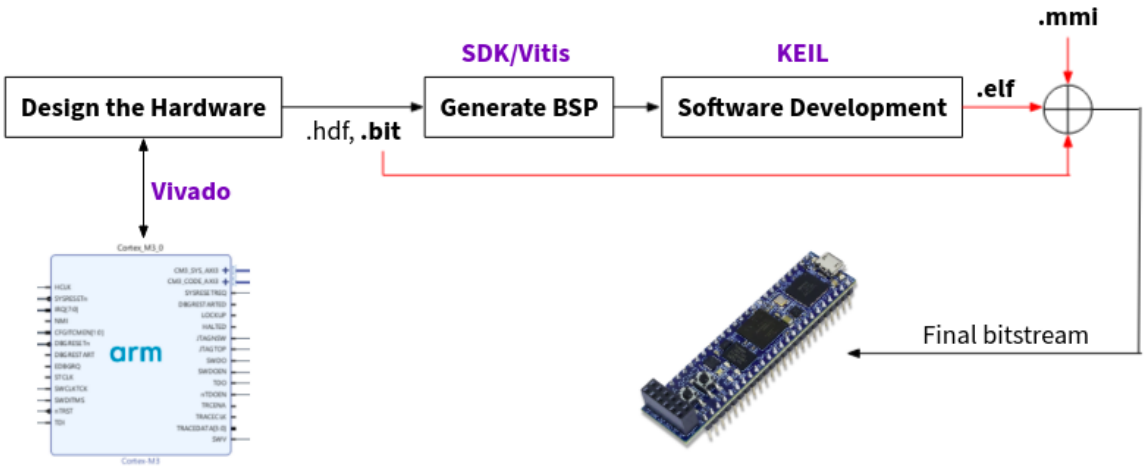


Figure 4.3: Implementation workflow for ARM Cortex-M IP

Once the bitsream is obtained, the next step is to generate the board support package (**BSP**) for the hardware platform. BSP is a set of files that provide low-level drivers for all the hardware. This allows for a consistent interface between

high-level software projects, and the low-level hardware drivers. This package will incorporate the memory map, for example, extracting the locations as consistently named variables. We would be using Xilinx **SDK 2019.1** for this purpose. For the software development, we would use the ARM **Keil-MDK 5** environment which is a software development solution for Arm-based microcontrollers. Although the software development route through Keil is a bit painful, we don't have any other options as these soft IPs are currently not supported by Vitis/SDK.

Once the application project is built successfully, we would obtain an **.elf** file. This, with the help of .mmi file, is used to update the program memory contents inside the bitstream. This method takes less than a minute, which is orders of magnitude faster than regenerating a bitfile. Finally, this updated bitstream is downloaded on the FPGA board. More details regarding the hardware and software configuration would be discussed in the further sections.

4.2 Hardware Configuration & Bitstream Generation

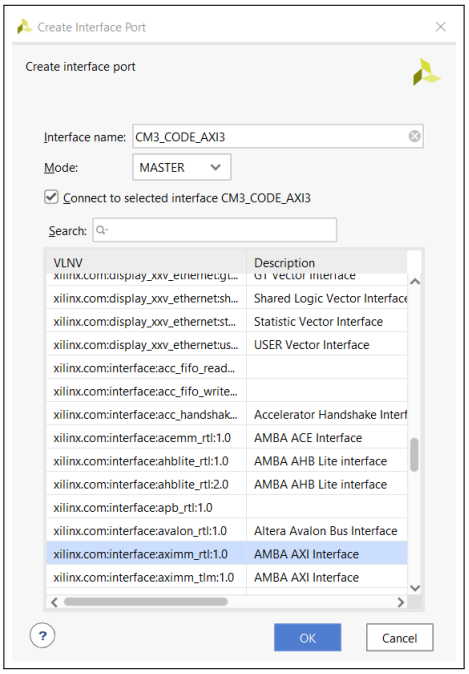
Vivado Design Suite is a software suite produced by Xilinx for synthesis and analysis of HDL designs. It allows us to synthesize and implement the hardware from a block level design using numerous pre-compiled Xilinx IPs and define interconnections among them, which drastically reduces the development time and time-to-market. FPGA is required to be configured with an appropriate hardware design which consists of Cortex-M3 IP as the CPU. The hardware and software package for this IP can be downloaded from the ARM designstart website.

The package comes up with an example vivado design project targeting Arty A7 and S7 boards. Users can modify this example project directly for their own target FPGA board. However, this process can be a bit longer and is prone to errors. Thus, we would create a new project in Vivado and define our design from scratch. Before we create the block deisgn, the Arm IP Integrator (IPI) repository for Cortex-M3 must be added to the list of Vivado IP repositories. This makes the processor available in any new designs. More details regarding the procedure and project initialization are mentioned in the *<package >/docs/arm_cortex_m3_designstart_fpga.../.pdf* doc. Users are requested to configure the project by following instructions till section 2.4.

Next, we can create a new block design in vivado and add the Cortex-M3 IP. Double clicking on this IP opens up the configuration window, where we can set various parameters such as the number of interrupts, debug level, size of ITCM (instruction memory) and DTCM (data memory), etc. Since the TFLite application would be big-

ger, we can set the ICTM size as 128KB. The DTCM size is proportional to the model size, however 64KB is a good starting value. Three of the most important external ports in the Cortex-M3 IP are listed below along with their usage and configuration.

1. **CM3_SYS_AXI3**: A memory-mapped master port which is used to access the external peripherals using the AXI Interconnect. Vivado would automatically connect this port while running the *connection automation*. Thus, we are not required to make any manual changes.
2. **CFGITCMEN[1:0]**: This input signal is the Instruction Tightly Coupled Memory (ITCM) alias enable. Bit [1] sets the upper alias enable line and bit [0] sets the lower alias enable line.
   If CFGITCMEN[1] is set, then the internal RAM ITCM is mapped to the upper address alias in the memory map.
   If CFGITCMEN[0] is set, then the internal RAM ITCM is mapped to the lower address alias in the memory map. Right-click the CFGITCMEN[1:0] input of the Cortex-M3 IP and select *Create Port*; accept pre-filled settings and click OK. Later, this would be assigned the constant value **1** in the top-level HDL code.
3. **CM3_CODE_AXI3**: The processor instruction fetches that are to memory addresses that are not aliased to the ITCM is output on the CM3_CODE_AXI3 port. Select this port in the Cortex IP, then right-click in the design window and select *Create Interface Port*; keep the pre-filled values and click OK.



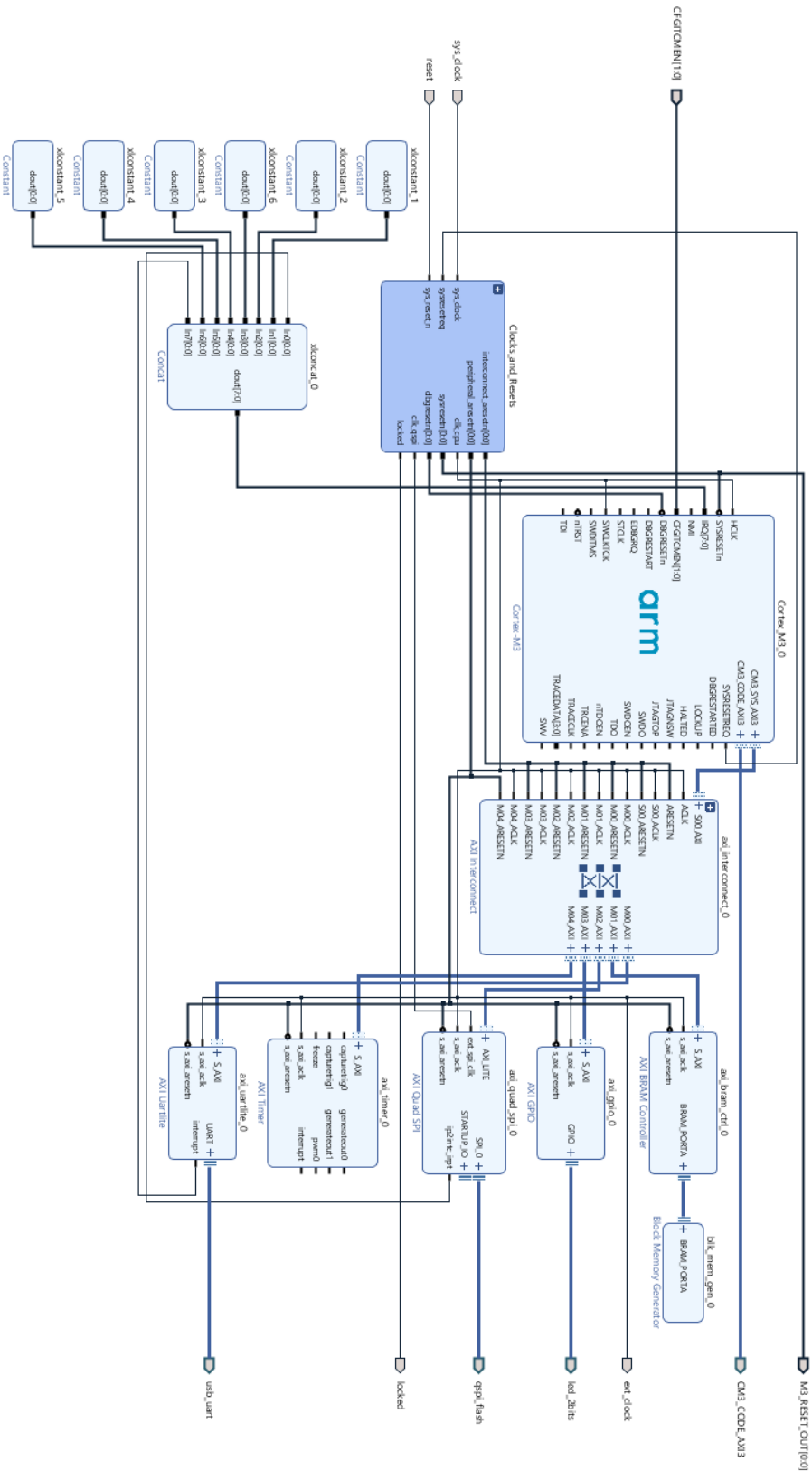


Figure 4.4: Block Design for using Cortex-M3 IP on CMOD A7-35T board

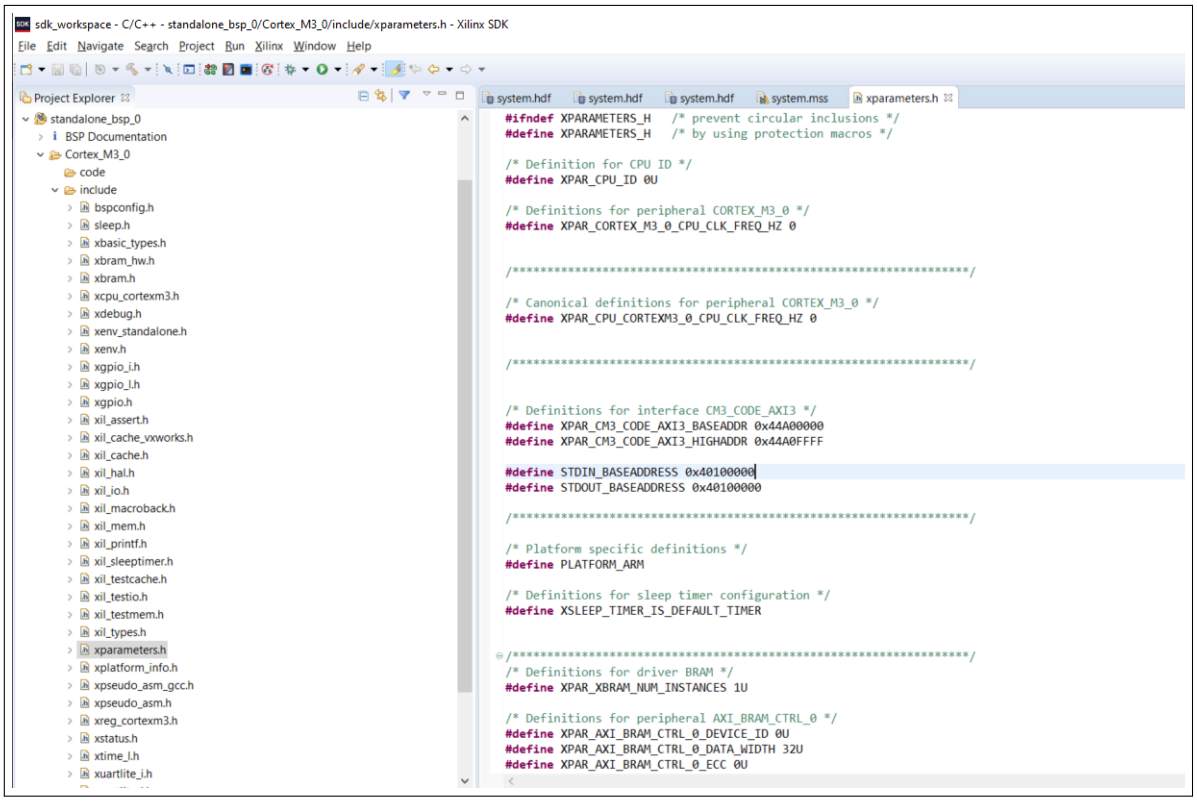
The interrupt port (**IRQ**) of Cortex IP can also be assigned a value of 0 using the constant blocks. Now, we can add and configure the external peripherals in our block design. For example, we can add the AXI Uartlite IP for enabling the UART communication, AXI GPIO for using the external LEDs, AXI Timer for using the timer module, etc. We also require clocks and reset to drive different components in the design. The *Block Automation* feature in vivado automatically connects these memory-mapped peripherals with the processor IP using an AXI interconnect. The final block design is shown in Figure 4.4.

The final step is to specify the top-level HDL file which binds our entire design. Go to the *Sources* tab in the block design window, right-click on the block design file (.bd) under the Design Sources, and select *Create HDL Wrapper*. This would generate a top-level verilog/vhdl file. If you are working with the CMOD A7-35T board, then replace the contents of this top-level file with the contents in cmod_a735t_arm.v. Otherwise, this top-level file can be modified similarly for other boards. Few other external ports (M3_RESET_OUT, locked, etc.) are also created in this design for debugging purpose. However, they can be removed and corresponding changes must be reflected in the top-level HDL file. Also, add the appropriate hardware constraint file (.xdc) for the specific board in the project. Finally, we can generate the **bitstream** (.bit) for our hardware design. Make sure that there are no errors encountered during this process.

4.3 Generation of BSP (Board Support Package)

Once the bitstream is generated successfully, the hardware can be exported to generate the BSP files. Xilinx Software Development Kit (SDK) is an eclipse-based Integrated Development Environment (IDE) for development of embedded software applications targeted towards Xilinx embedded processors. In the current development flow, this is used to generate the required BSP files. Follow the steps mentioned in the *<package>/docs/arm_cortex_m3_designstart_fpga.../.pdf* doc for the BSP generation. The Standalone OS version used in the current design is 6.7. However, the package files can be modified to support even higher versions!
If you are using *stdin* and *stout* or any print statement in your C/C++ application project, then make sure that these are set as *axi_uartlite_0* in the BSP settings. Failure to do so would possibly result in a non-functional UART peripheral. Also, check whether the *STDIN_BASEADDRESS* and *STDOUT_BASEADDRESS* in the *xparame-*

ters.h file are correctly set to the UART peripheral base address.
Lastly, we need to manually copy 2 files into the generated BSP location because of the differences between the SDK and Arm Keil-MDK. Copy the files *xpseudo_asm_rcvt.h* and *xpseudo_asm_rcvt.c* from - *<package>/vivado/Arm_sw_respository/CortexM/bsp/standalone_v6_7/src/arm/cortexm3/armcc* to *<sdk_workspace>/standalone_bsp_0/CORTEX_M3_0/include*. The BSP is complete and is now ready for compilation.



4.4 TFLite Software Development

The example *hello_world* project created during the TFLite build process in section 3.1 also generates the project files for the Keil environment. It contains the *µVision* project (.uvprojx) along with the other TFLite source files. The idea is to modify this example project and make it compatible with the FPGA platform.

Modify Target Configuration:
Before we proceed, we need to copy few files/folders to our current working direc-

tory of Keil project. First, copy the ***cmsis*** folder located under the *software* section of the Cortex-M3 package. Second, copy the bsp folder ***standalone_bsp_0*** which was generated in the previous section. Lastly, copy the *<Cortex-M3 package>/software/m3_for_arty_a7/main/m3_for_arty.h* file into the current working directory.
Open the example Keil project *(keil_project.uvprojx)*. In the Project window, right click on the *hello_world* and select *Options for Target 'hello_world'*. This would open up the target configuration window. Following are the changes that we are required to perform in the target configuration -

1. **Device**: In the Device tab, select the device as ARM Cortex M3 (ARMCM3).

2. **Target**: In the Target tab, the size of IROM1 and IRAM1 needs to be changed. Since our ITCM memory is 128KB, we can set the size of IROM1 as 0x20000. Similarly, size of IRAM1 is set as 0x10000.

3. **User**: Keil generates a .axf file after the application project is built successfully. This .axf file needs to be converted to hex format for BRAM initialization, and to .elf format for bitstream generation. A windows batch file *make_hex_a7.bat* serves this purpose and should be automatically run post-build. This file is available in the downloaded Cortex-M3 package under the *software/Build_Keil* section. However, few changes are required to be made in this file and an example make_hex_a7.bat is provided.
   In the user tab, check the *Run #1* box under the *After Build/Rebuild* option in Command Items. Also, select the corresponding *User Command* and specify the path to the *make_hex_a7.bat* file.

4. **C/C++ (AC6)**: In this tab, we can define the preprocessor symbols and the include file directories. In the *Define* section of *Preprocessor Symbols*, enter the line **ARTY_CM3 CORTEX_M3**. Next, select the *Include Paths* and open the browse window. Here, we need to add few include directories which correspond to the FPGA hardware and ARM platform as shown in Figure 4.5.

5. **Linker**: In this tab, check the *Use Memory Layout from Target Dialog* box. The *Scatter File* section should have automatically filled up. In case it is empty, try deselecting and selecting this box again. Figure 4.6 shows the tab after the required changes. Click OK.

The target configuration is completed. Now, we need to add the source files and manage the run-time environment settings.

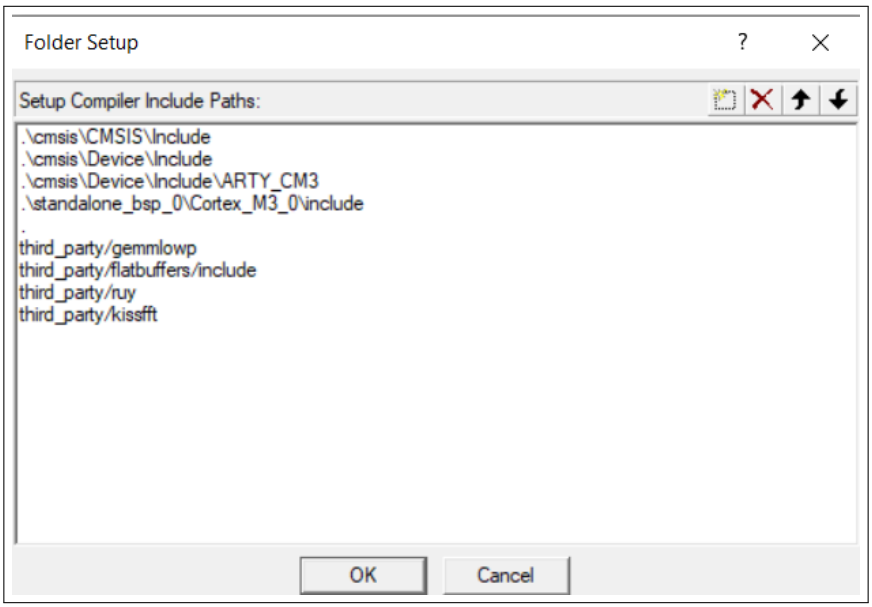


Figure 4.5: Include directories for the Keil project

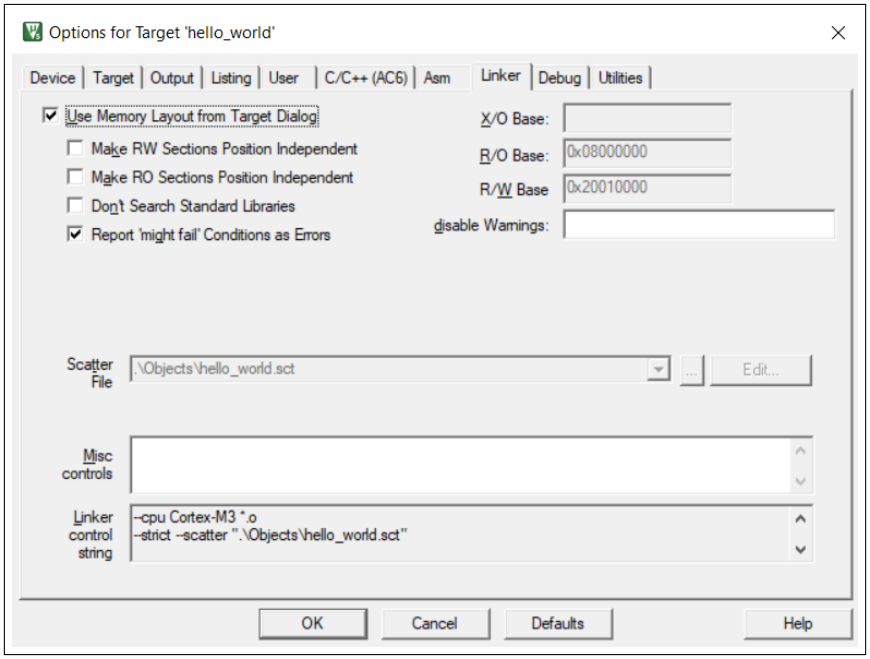


Figure 4.6: Linker tab after required changes

Add Source Files:

In the Project window, right click on the *hello_world* and select *Manage Project Items*. Here, we need to select various source files under the *Project Items* section. The

TFLite source files are already added under the group named ***Source***. Create a new group named ***CMSIS*** and add the following 2 files under it -

1. *cmsis/Device/Source/ARTY_CM3/system_ARTY_CM3.c*
2. *cmsis/Device/Source/ARTY_CM3/ARM/startup_ARTY_CM3.s*

Next, we need to add the BSP specific files. Create a new group named ***Standalone_v6_7*** and add the following files under it -

1. *standalone_bsp_0/Cortex_M3_0/libsrc/standalone_v6_7/src/print.c*
2. *standalone_bsp_0/Cortex_M3_0/libsrc/standalone_v6_7/src/outbyte.c*
3. *standalone_bsp_0/Cortex_M3_0/libsrc/standalone_v6_7/src/xil_assert.c*

Next, we need to add source files for the peripherals. For example, create a new group named ***Xilinx_UART*** and add all the files under the folder *standalone_bsp_0/Cortex_M3_0/libsrc/uartlite_v3_2/src*. The exact path would depend on the version of the peripherals. Repeat this procedure for all the required peripherals in the application project.

Manage Run-Time Environment:

Go to **Project >Manage >Run-Time Environment**. Under the *CMSIS* section, select the checkbox for the **CORE** option. Click OK. The final project items are shown in Figure 4.7. Now, we are ready to write the application code for our TFLite project.

Writing the Application code:

We need to modify 2 files - *main.cc* (application code) and *model.cc* (contains the model parameters/flatbuffer) under the group *Source*. We can simply replace the contents of the flatbuffer array in *model.cc* with the corresponding contents of our trained model file (*model.h*). In the file *main.cc*, we need to include the headers as follows -

```
#include "m3_for_arty.h"   // Platform specific
#include "xparameters.h"   // Identifying peripherals
#include "xuartlite.h"     // UART IP header file
#include "xil_printf.h"    // Using print statements
#include "xtmrctr.h"       // Timer IP header file
```

```
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"

// Include Model file
#include "tensorflow/lite/micro/examples/hello_world/model.h"
```

Rest of the application code is same as explained in the section 3.2, with minor changes in the definition and initialization of the peripherals. For example, to initialize the UART peripheral, we write the following code -

```
// Declare UART IP instance
static XUartLite UART_instance;

// Initialize the UART peripheral
XUartLite_Initialize(&UART_instance, XPAR_AXI_UARTLITE_0_DEVICE_ID);
```

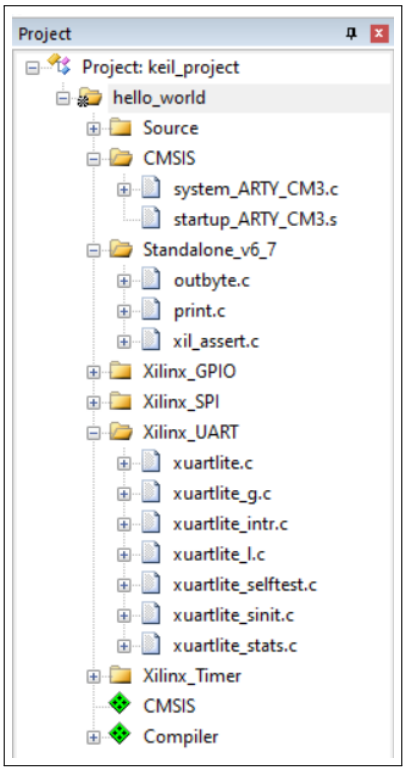


Figure 4.7: Keil Project Items

An example application code is provided at main.cc. Before we build the target, we are required to do some changes in few of the source files as mentioned below -

1. **flatbuffers.h**: This file is present under the *Source* group. We are required to find and comment the following section of code in this file -

```
//extern volatile __attribute__((weak)) const char *
    flatbuffer_version_string;
//volatile __attribute__((weak)) const char *
    flatbuffer_version_string =
//  "FlatBuffers "
//  FLATBUFFERS_STRING(FLATBUFFERS_VERSION_MAJOR) "."
//  FLATBUFFERS_STRING(FLATBUFFERS_VERSION_MINOR) "."
//  FLATBUFFERS_STRING(FLATBUFFERS_VERSION_REVISION);
```

   Failure to do so would result in an error like -
   ERROR: flatbuffers::flatbuffer_version_string Multiply defined Global Symbol

2. **xpseudo_asm_gcc.h**: This file is located in *standalone_bsp_0/Cortex_M3_0/include*. The *str(adr, val)* function defined in this file overlaps with one of the function defined in the TFLite source files. Thus, this function must be commented out in the file *xpseudo_asm_gcc.h*.

```
    /*#define str(adr, val)      __asm__ __volatile__(\
        "str     %0,[%1]\n"\
        : : "r" (val), "r" (adr)\
      )*/
```

3. **debug_log.cc**: This file is located under the *Source* group. We must modify the function *DebugLog* as follows. This function is later redefined in *main.cc* as shown in the example file.

```
extern "C" void __attribute__ ((weak)) DebugLog(const char* s){
   // To be implemented by user
}
```

Finally, we can build the target and obtain the **.elf** file. Note that *make_hex_a7.bat* is automatically run post-build and the generated files (.hex and .elf) are located in the current working directory of the Keil project.

4.5 Running the Application

As discussed earlier, we require a .mmi file to identify the program memory locations inside the bitstream. We can either manually write this file which is prone to human error, or we can use a helper script ***make_mmi_file.tcl*** to automatically generate this file. Also, to automate the process of merging the .elf file with the bitstream, we have a batch file ***make_prog_files.bat***. Both of these scripts come along with the Cortex-M3 package and are located under the hardware section. We need to copy 4 files - *make_mmi_file.tcl*, *make_prog_files.bat*, *make_prog_files.tcl* and the generated .elf file in the previous section (***bram_a7.elf***) into the vivado project directory, and make few changes inside them as discussed further.

Generating .mmi file:
Open the *make_mmi_file.tcl* file and edit the *set part* line to the correct part for your board. For CMOD A7-35T, the part is *xc7a35tcpg236-1*. In Vivado, click "Open Implemented Design", then in the TCL console, navigate to the project directory and run: *$ source make_mmi_file.tcl*. This would generate a MMI file named **m3.mmi**.

Update the Bitstream:
We need to edit the file *make_prog_files.tcl*. Open it and make the following changes -

1. Rename *source_bit_file* to *"./<runs folder >/impl_1/<Bit file name >"*
2. Rename *output_bit_file* to *"final.bit"*

We can also modify the names for MMI and ELF file, however this should be consistent with the previous sections. Also, add the path of bin folder for both Keil and Vivado in the system and user path environment variables, respectively. Finally, since all the required files are now available (*m3.mmi*, *bram_a7.elf* and bitstream), we can run the batch file *make_prog_files.bat*. This would generate the final updated bitstream which can later be downloaded on the FPGA board. Done!

4.6 Challenges

Since the current software development uses the Keil environment, this poses various challenges in the development process as follows -

1. **License Issue**: The free version of Keil-MDK has various limitations. For example, the programs that generate more than **32 KB** of code and data will not compile, assemble, or link. TFLite applications are generally much bigger (>100KB) and thus, they can't be compiled using the free version. For this report, we are using a 30-day free license of Keil.

2. **Complex process**: The development through Keil is a bit difficult as the IDE environment is non-intuitive. Also, it becomes very difficult to import and modify various source or header files inside the project. Navigation across the project is also non-trivial.

Working with **Eclipse**-based IDEs such as Vitis, SDK, etc. is much easier and faster. Also, there is no limitation in terms of the code size. We have tried our best to develop the present software using the Vitis/SDK environment, however, the board doesn't responds to any of our changes. I have even worked with low-level linker, startup files and have tried to modify the project for a GCC environment. The ARM-MDK uses the ARM compiler, whereas the SDK/Vitis environment uses the GCC toolchain. Although, I have also tried to use the ARM CC toolchain to compile the code in Vitis/SDK, yet there was no success. I am able to compile the code and generate the .elf file, however the final bitstream is not working. I think that the issue lies with the compiler and the linker flags.

This chapter was centered around the implementation of TFLite applications on the ARM Soft-IPs, which also allows the resource-constrained FPGA boards such as CMOD A7-35T to execute these applications. We can also use this process to develop and deploy an ARM-based software and thus, this can be used for the software prototyping and testing without the need of an actual hardcore ARM processor. Although the current IP has few limitations, it gives us a lot of opportunities to develop various interesting applications.

Chapter 5

Implementation on MicroBlaze

The last chapter introduced the advantages of using an FPGA for developing machine-learning applications and the implementation on a ARM soft CPU. Since the TFLite library that we created during the build process in section 3.1 is target-neutral, it can be easily imported and used with the other target architectures. One such platform is the Xilinx's soft-core processor, MicroBlaze which also overcomes many of the existing limitations of the ARM soft-IPs and the development process. Rest of the chapter focuses on the implementation of TFLite applications on MicroBlaze.

5.1 Introduction to MicroBlaze

The MicroBlaze CPU is a family of drop-in, modifiable preset 32-bit/64-bit Harvard RISC microprocessor architecture optimized for implementation in Xilinx FPGAs [9]. It has 32-bit instruction set and general purpose registers with 32-bit address bus, extensible to 64 bits. MicroBlaze is a highly configurable processor. The basic design can be user configured with more advanced features such as barrel shifter, divider, multiplier, single precision floating-point unit (FPU), instruction and data caches, exception handling, debug logic, Fast Simplex Link (FSL) interfaces and others. With few exceptions, the MicroBlaze can issue a new instruction every cycle, maintaining single-cycle throughput under most circumstances.
This flexibility allows the user to balance the required performance of the target application against the logic area cost of the soft processor. Although, the overall throughput of MicroBlaze is substantially less than a comparable hard CPU core (such as the ARM Cortex-A9 in the Zynq), the developer can make the appropriate design trade-offs for a specific set of host hardware and application software requirements. Since the MicroBlaze is a soft-core microprocessor, any optional fea-

tures which are not used will not be implemented and thus, will not take up any of the FPGA resources. Figure 5.1 shows a functional block diagram of the MicroBlaze core. MicroBlaze can be used as a stand-alone processor in all Xilinx FPGAs or as a co-processor in a various SoC systems such as Zynq. It is commonly used in one of three preset configurations as shown in the table below: a simple micro-controller running bare-metal applications; a real-time processor featuring cache and a memory protection unit interfacing to tightly coupled on-chip memory running FreeRTOS; and finally, an application processor with a memory management unit running Linux. The table below shows the performance and utilization estimates for these configurations -

	Microcontroller	**Real-Time**	**Application**
Max Clock freq. (MHz)	204	172	146
Logic Cells	1900	4000	7000

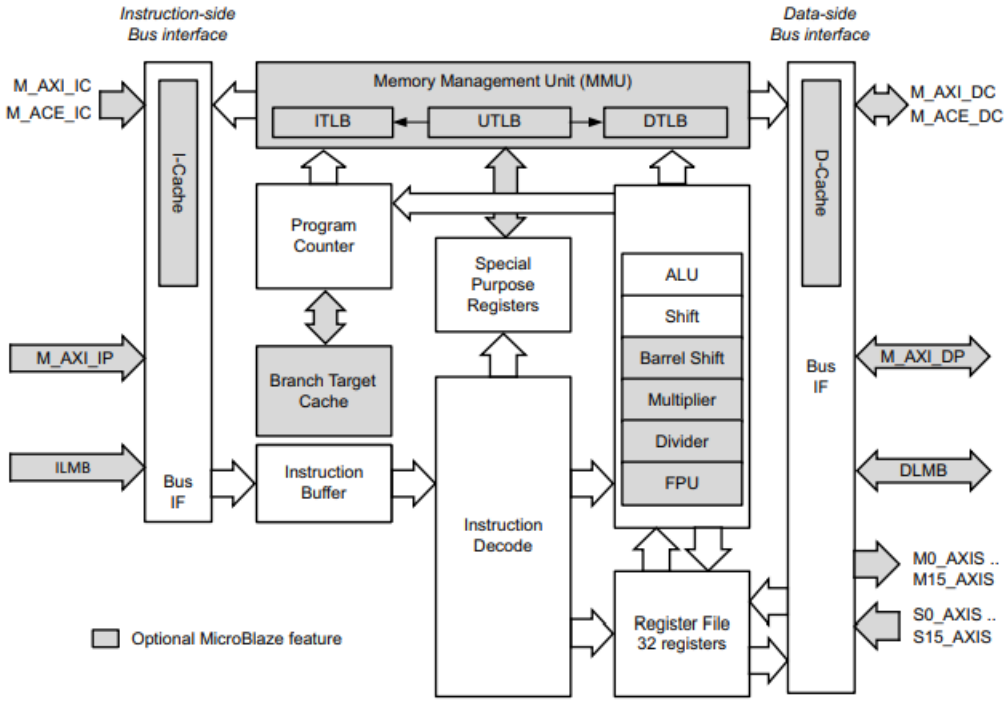


Figure 5.1: MicroBlaze Core Block Diagram (Source [10])

Since we require to run a bare-metal TFLite application, we would be using the Microcontroller variant. More details regarding the hardware and software configuration would be discussed in the further sections.

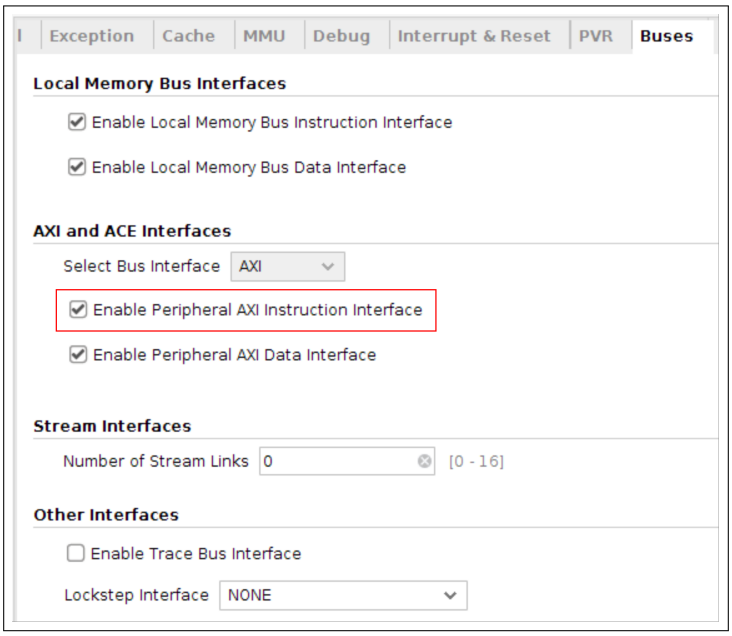
5.2 Hardware Configuration & Bitstream Generation

The hardware configuration would be similar to section 4.2. However, in this case, we would be using the MicroBlaze soft IP as the CPU. We are not required to download any IP package as microblaze is fully supported by Vivado and SDK/Vitis.

Open the Vivado Design suite and create a new project for the CMOD A7-35T board. Further, we need to create a new block design and add the MicroBlaze IP. Double clicking on this IP opens up the configuration window. Since, we want the most performance-optimized version of the MicroBlaze, we need to enable various options in the *General* tab including the Barrel shifter, FPU (Extended), Integer multiplier, etc. as shown in the Figure 5.2. A few more changes in the microblaze configuration are listed as follows. Note that the following changes are specific to the CMOD A7-35T board. For example, we may not require an instruction cache if enough amount of BRAM is available for code memory.

Enable AXI Instruction Interface:

The TFLite applications would not fit in the limited amount of BRAM memory (225KB) available on the CMOD A7-35T board. Thus, we require some external memory device to hold the .text section of our code memory. Fortunately, we have an external SRAM (512KB) available on the board. To access the instructions from this external memory, MicroBlaze requires a peripheral AXI instruction interface and thus, this needs to be enabled as shown in the following figure.



Enable Instruction & Data Cache:

External memory devices are slower and would result in higher latency while fetching instructions. Thus, we should enable the **Instruction cache** of microblaze. It uses the internal BRAM as the cache, thus the access latency is close to 1 cycle. The base and high address in the cache must be carefully set to that of the external SRAM.

Data cache plays a vital role while accelerating the kernels using the Custom IPs. The ILMB and DLMB memory are private to the microblaze and hence, their data can't be accessed by any other external peripheral or IP. However, we can add a shared memory location (BRAM) which can be accessed by both microblaze and the custom IPs. This step should be applicable for all the FPGA boards which require custom IPs for the acceleration.

First, we add the AXI BRAM Controller IP and a Block Memory Generator(BRAM) in the block design. Running connection automation should automatically connect this BRAM with the AXI BRAM controller. Further, this AXI BRAM controller IP is manually connected with the *M_AXI_DC* port of the microblaze. Note that the base and high address in the data cache of microblaze must be carefully set to that of this BRAM address. The microblaze cache configuration is shown in Figure 5.3.

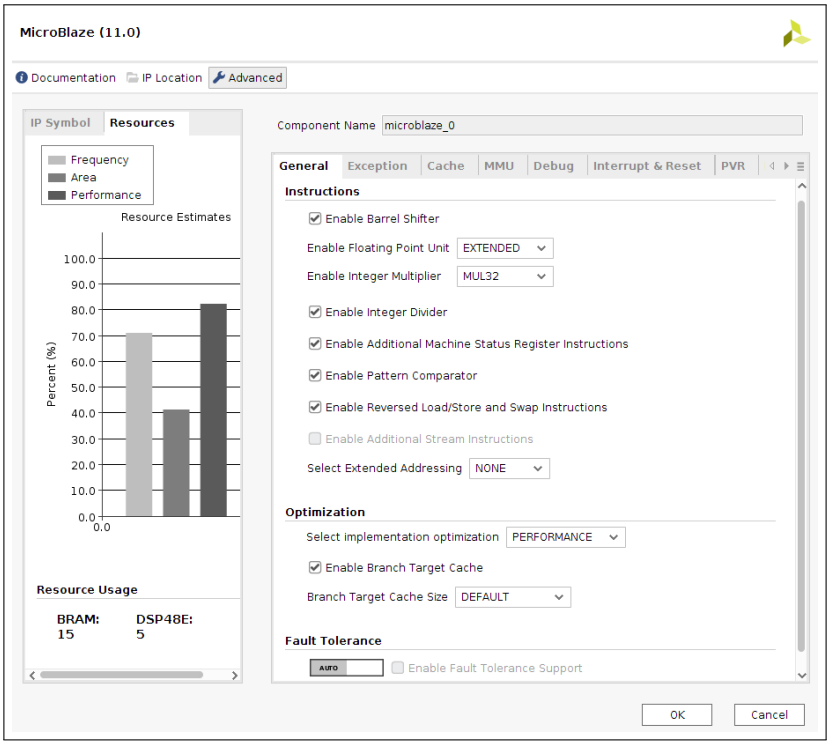


Figure 5.2: MicroBlaze configuration

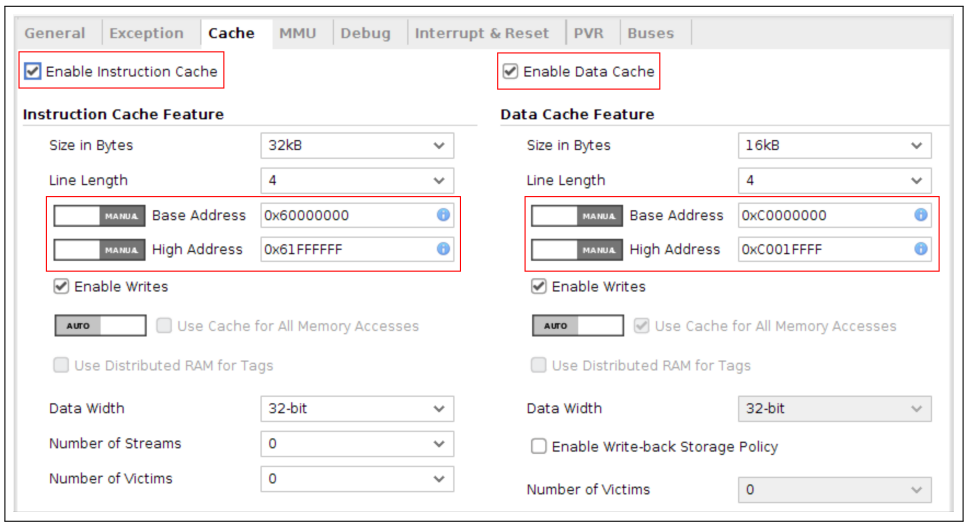


Figure 5.3: Cache configuration in MicroBlaze

Till here, the microblaze connections should look similar to the Figure 5.4. The above changes would increase both the **performance** and the **resource-utilization** of the processor. Finally, we can add the required peripheral IPs in our block design. For this discussion, I have created a basic design which consists of the external SRAM and UART peripheral. Running the *block* and *connection automation* in vivado would automatically connect these to the microblaze using the AXI Interconnect bridge. The *M_AXI_DP* (peripheral data port), *M_AXI_IP* (instruction port) and *M_AXI_IC* (instruction cache) are also connected to the AXI interconnect. The final block diagram is shown in Figure 5.5.

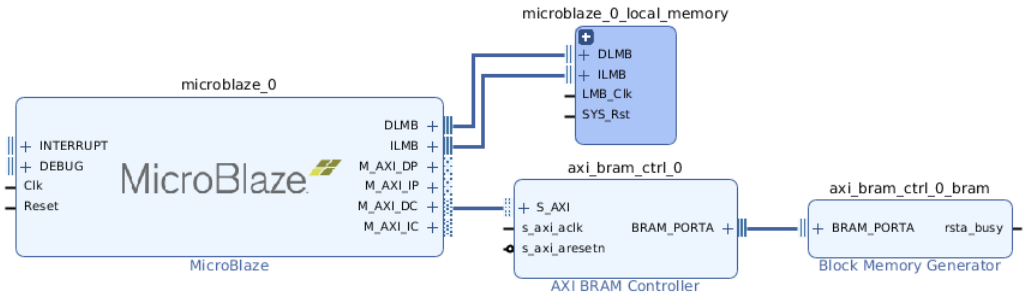


Figure 5.4: Data cache connection in MicroBlaze

Further, we can create the top-level HDL wrapper for our design and generate the bitstream. Next, we would develop the software using Xilinx SDK for running the TFLite applications on microblaze.

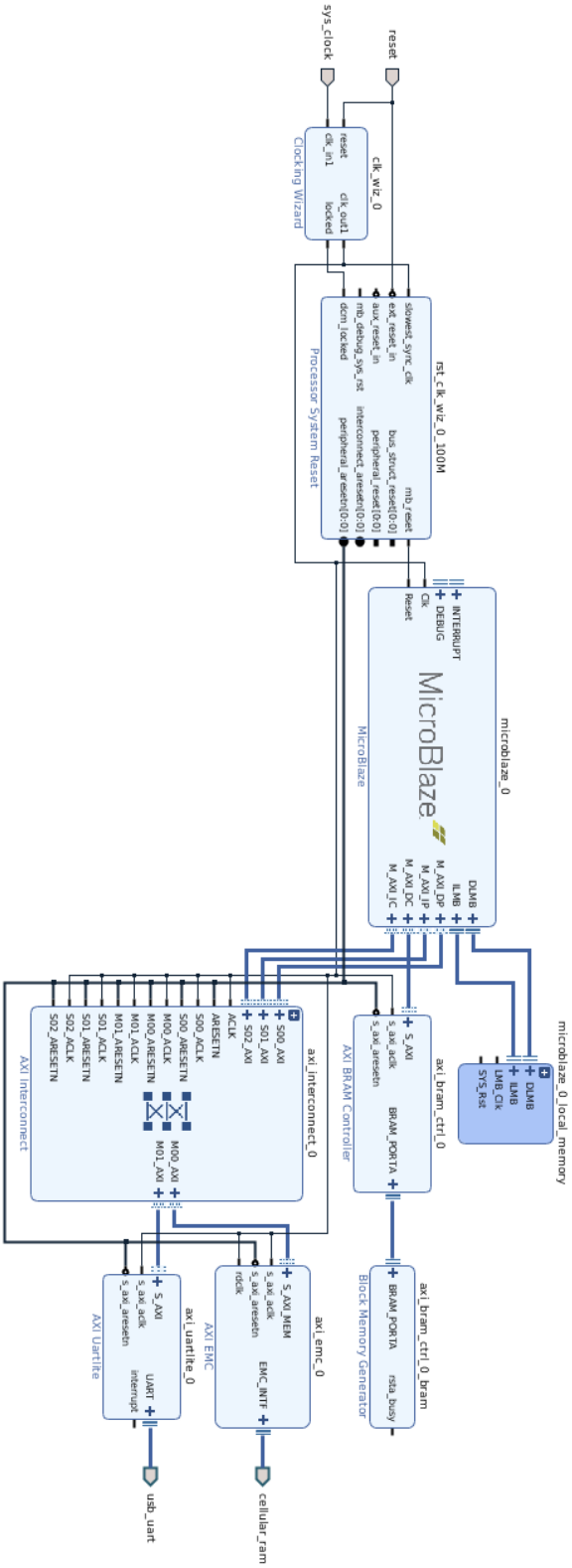


Figure 5.5: Block Design for MicroBlaze

5.3 TFLite Software Development

Since MicroBlaze is fully supported by the SDK/Vitis, we can directly create an application project for our hardware platform. This would auto-generate the required board support package (BSP) files. The implementation workflow is as follows -

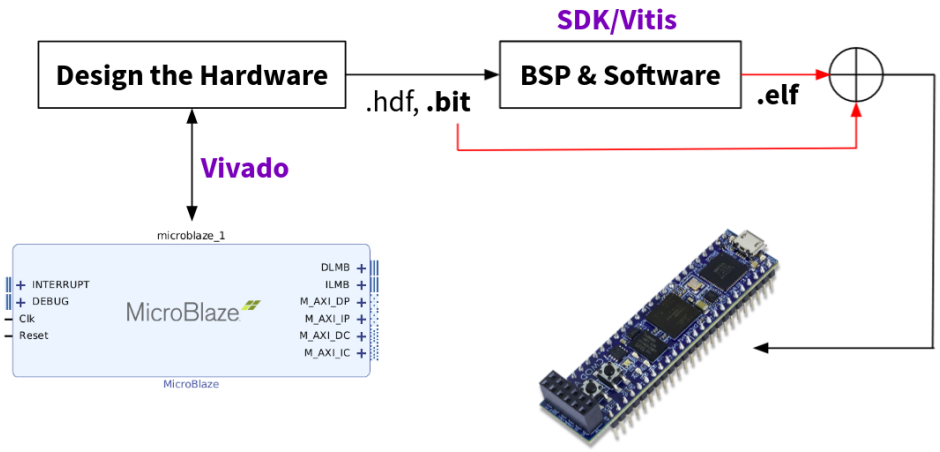


Figure 5.6: Implementation workflow for MicroBlaze application development

Create a C++ application project in SDK/Vitis using the exported hardware definition file. Since SDK/Vitis is an eclipse-based IDE, this project would be configured similar to section 3.2. Copy the flatbuffer file *model.h* into *<sdk_project>/src*. Also, create a new folder *tensorflow_lite* under *<sdk_project>* and copy the TFLite source folders (*tensorflow* & *third_party*) into this. A sample file structure is shown in Figure 5.7. Next, we must specify the include directories in our SDK project as shown in Figure 5.8. Source location for TFLite is already specified due to the file structure in SDK project (no changes required). Don't forget to delete the DSP and CMSIS-NN related files as these aren't supported by microblaze. Finally, we can write our TFLite application code similar to section 3.2, with minor changes in the definition and initialization of the peripherals (discussed in previous chapter). An example application code is provided at main.cc.

Before we compile our application, we need to generate the **Linker** file for mapping different code sections at appropriate memory locations. Select the application project and go to **Xilinx >Generate linker script**. Specify the location of Code sections (SRAM), Data and Heap sections(BRAM). Click Generate and compile the application to obtain the .elf file. To reduce the code memory, the Optimization setting in C++ build can be selected as *Optimize for size (**-Os**)*.

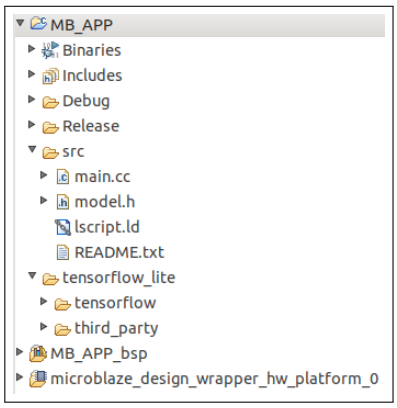


Figure 5.7: SDK Project file structure

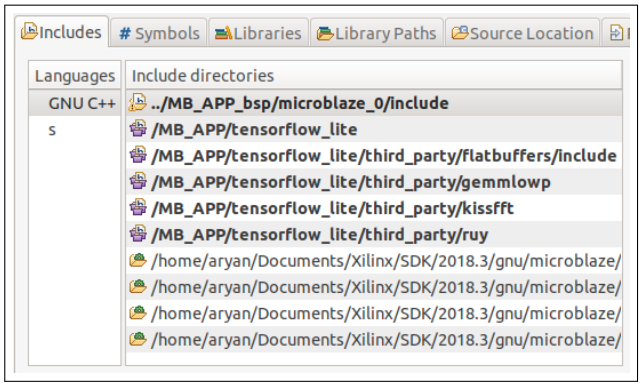


Figure 5.8: SDK Include directories

5.4 Acceleration using Custom IPs

The FPGA fabric allows us to develop custom hardware accelerators for accelerating our application and improving the run-time. We can offload few data-intensive operations from the microblaze and execute them via the implemented custom IPs. One such operation could be the Fully connected layer operation as discussed in section 3.3.1. In this example, we intend to accelerate the matrix multiplication stage in the fully connected layer using a custom IP. The first step would be to synthesize and implement this IP, and later integrate it with the current hardware design.

5.4.1 Creation of Custom IP using Vivado HLS

The Xilinx Vivado High-Level Synthesis (HLS) tool transforms a C specification into a register transfer level (RTL) implementation which we can synthesize into a Xilinx FPGA. We can control the C-synthesis process using various optimization pragmas to create high-performance implementations. This includes pipelining loops, unrolling, etc. Thus, the hardware designers can work at a higher level of abstraction while creating high-performance hardware.

I have designed a custom IP which evaluates the dot product of 2 arbitrary-sized vectors, after adding a constant offset. This IP would be later used in the integer TFLite kernel. For accelerating the process and improving parallelism, the multiply and accumulate stage is implemented in the form of an ***Adder tree*** as shown in the Figure 5.9. The outermost loop is unrolled 8-times. This would highly increase the resource-utilization while improving the operation latency.

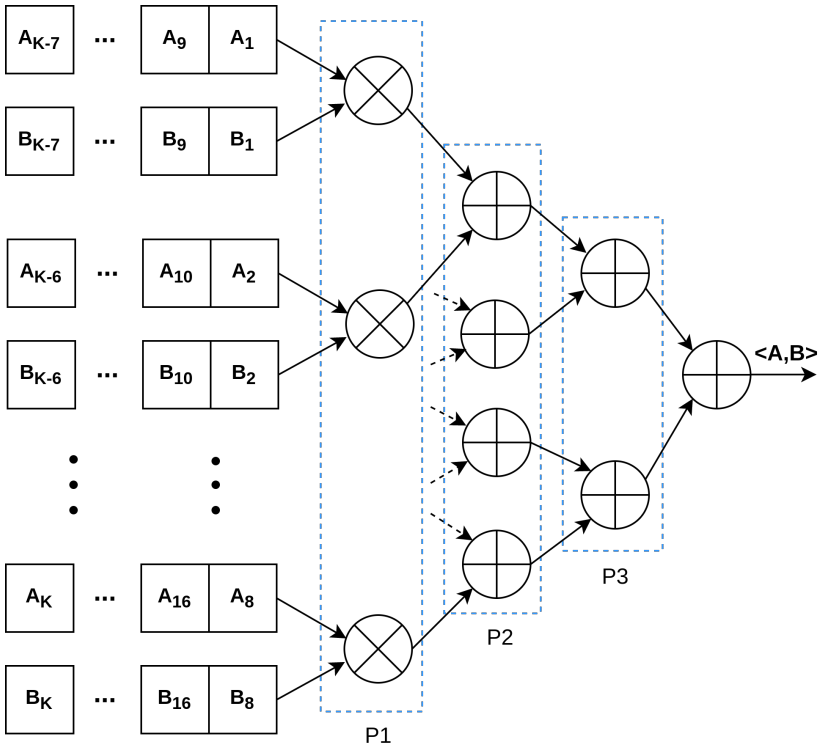


Figure 5.9: Multiply & Accumulate stage in the Custom IP

In HLS, the above functionality can be implemented as follows -

```
// inp -> Input vector
// wt -> Weight vector

for(int j=0; j<8; j++){
  #pragma HLS UNROLL factor=8
  mult[j] = inp[j] * wt[j];
}
for(int j=0; j<4; j++){
  add1_arr[j] = mult[2*j] + mult[2*j + 1];
}
for(int j=0; j<2; j++){
  add2_arr[j] = add1_arr[2*j] + add1_arr[2*j + 1];
}
acc += add2_arr[0] + add2_arr[1];
```

These individual arrays are partitioned in memory, so that multiple elements can be accessed simultaneously. Using HLS, we can easily define the type of interface for

the individual operands using the pragma *HLS INTERFACE*. The 2 input vectors are read from a memory-mapped AXI interface (***m_axi***). Thus, we can treat these inputs as array pointers. Rest of the inputs (input offset, weight offset and size of vector) are declared as AXI4-Lite interface (***s_axilite***) as shown below -

```
int FullyConnected(signed char* input,
                   signed char* weight,
                   int input_offset,
                   int weight_offset,
                   int accum_depth)
{
#pragma HLS INTERFACE m_axi depth=200 port=input offset=slave
    bundle=MASTER_BUS
#pragma HLS INTERFACE m_axi depth=200 port=weight offset=slave
    bundle=MASTER_BUS
#pragma HLS INTERFACE s_axilite port=input_offset bundle=CRTL_BUS
#pragma HLS INTERFACE s_axilite port=weight_offset bundle=CRTL_BUS
#pragma HLS INTERFACE s_axilite port=accum_depth bundle=CRTL_BUS
#pragma HLS INTERFACE s_axilite port=return bundle=CRTL_BUS
 ....
}
```

The **bottleneck** here is the memory-access! We can't read multiple elements (input and weight vector) from the data memory of microblaze simultaneously as it has only a single read port. Nevertheless, rest of the operations are now much faster. The complete HLS code is provided at fullyconnected.cpp.
Once the design is finalized, we can synthesize and implement this Custom IP. Timing violations must be checked, if any. Table 5.1 shows the resource-utilization for this IP. Finally, we can export the RTL to integrate it in our Vivado project.

BRAM_36K	DSP	Flip-Flop	LUT
1	24	2661	3988

Table 5.1: Resource utilization for the Custom IP (Fully Connected)

5.4.2 Integration in Vivado

Open the microblaze vivado project and add the IP repository of the fully connected module in the project settings. This allows vivado to identify and import our custom

IP in the block design. Now, add this Custom IP to our block design.

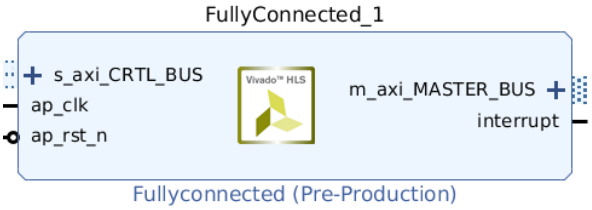
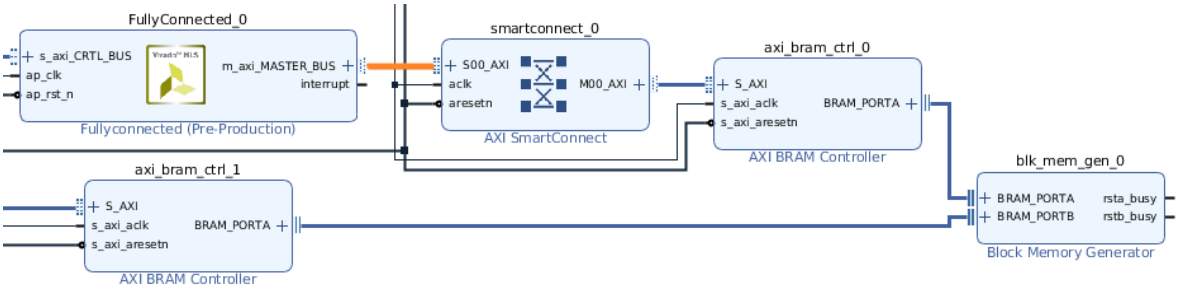


Figure 5.10: Fully Connected IP

Since we want to share and access the data memory of microblaze, we need to increase the BRAM ports of the microblaze data memory. Double click on the corresponding BRAM and select the *Memory Type* as *True Dual Port RAM*. Thus, one of the port will be accessed by the microblaze and the other would be accessed by various custom IPs in the design.
To connect the fully connected IP with the data BRAM, we need to include 2 IPs in our block design: AXI SmartConnect and AXI BRAM Controller. The memory-mapped interface of the custom IP would be connected to the slave interface of AXI SmartConnect, whose master port is further connected with the AXI BRAM controller. This is again connected with the other BRAM port. The final connection should appear as follows -



If the design consists of more than one Custom IP, then we can simply increase the number of slave ports of the AXI SmartConnect, and connect them with the individual custom IPs. More number of custom IPs would reduce the available data bandwidth and might slow down the data access. Once the design is validated, we can generate the bitstream and export the hardware.

5.4.3 Modifying the TFLite Kernel

The SDK project must be upgraded with the newly generated hardware description file in the previous section. Before using the custom IP in our application, we are first required to initialize it as follows -

```
#include "xparameters.h"

// Instance of the Custom IP
XFullyconnected xfc;
XFullyconnected_Config *xfc_cfg;

xfc_cfg = XFullyconnected_LookupConfig(XPAR_FULLYCONNECTED_0_DEVICE_ID);
if(xfc_cfg){
   int status = XFullyconnected_CfgInitialize(&xfc, xfc_cfg);
   if(status != XST_SUCCESS)
      return XST_FAILURE;
}
```

Now, we will make changes in the integer TFLite kernel which is located at - *<sdk_project>/tensorflow_lite/tensorflow/lite/kernels/internal/reference/integer_ops/* ***fully_connected.h***.

By default, the kernel code uses normal *for* loops for computation as follows -

```
for (int b = 0; b < batches; ++b) {
    for (int out_c = 0; out_c < output_depth; ++out_c) {
      int32_t acc = 0;
      for (int d = 0; d < accum_depth; ++d) {
        int32_t input_val = input_data[b * accum_depth + d];
        int32_t filter_val = filter_data[out_c * accum_depth + d];
        acc += (filter_val + filter_offset) * (input_val + input_offset);
      }
      if (bias_data) {
        acc += bias_data[out_c];}
      acc = MultiplyByQuantizedMultiplier(acc, output_multiplier,
          output_shift);
      acc += output_offset;
      acc = std::max(acc, output_activation_min);
      acc = std::min(acc, output_activation_max);
      output_data[out_c + output_depth * b] = static_cast<int8_t>(acc);
    }
}
```

The value of variable *batches* is generally 1. Thus, we can ignore this variable for simplifying the loop. Instead of iterating over each element, we can perform the above operation using our custom IP as follows -

```
// Set Inputs
XFullyconnected_Set_input_r(&xfc, (u32)input_data);
XFullyconnected_Set_input_offset(&xfc, input_offset);
XFullyconnected_Set_weight_offset(&xfc, filter_offset);
XFullyconnected_Set_accum_depth(&xfc, accum_depth);

for (int b = 0; b < batches; ++b) {
    for (int out_c = 0; out_c < output_depth; ++out_c) {

      XFullyconnected_Set_weight(&xfc, (u32)(filter_data + (out_c *
          accum_depth)));

      XFullyconnected_Start(&xfc);            // Start the computation
      while(!XFullyconnected_IsDone(&xfc)); // Wait till over

      int32_t acc = XFullyconnected_Get_return(&xfc);  // Get Output

      if (bias_data) {
        acc += bias_data[out_c];
      }
      acc = MultiplyByQuantizedMultiplier(acc, output_multiplier,
          output_shift);
      acc += output_offset;
      acc = std::max(acc, output_activation_min);
      acc = std::min(acc, output_activation_max);
      output_data[out_c + output_depth * b] = static_cast<int8_t>(acc);
    }
}
```

Thus, we can create our own custom IPs and integrate them further in the hardware design to accelerate the TFLite applications.

This chapter was centered around the implementation of TFLite applications on MicroBlaze, which is a highly configurable soft processor. It also sheds light on the usage and implementation of custom hardware accelerators which can drastically reduce the application run-time. An easier development process and high amount of flexibility makes the MicroBlaze favourable for developing TFLite applications on resource-constrained FPGAs.

Chapter 6

Performance Analysis

The previous chapters discussed about the implementation of TFLite framework on various platforms such as STM32 boards, ARM soft-IP and MicroBlaze. We can now compare the performance obtained on these platforms and the corresponding trade-offs. For this comparison, we require few benchmark models or applications via which we can compare the run-time latency. I have trained and tested 2 NN models: ***Sine*** (computes the sine value of a number), and the well known ***MNIST*** classification. The later model is also quantized to experiment with various optimization techniques. The resource-utilization in FPGAs are also reported.

6.1 Floating-precision Models

Sine Model:

A simple model which computes the sine value of an input number. This model has **321** parameters and 3 fully connected layer with the following architecture -

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                32
_________________________________________________________________
dense_1 (Dense)              (None, 16)                272
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 17
=================================================================
Total params: 321
Trainable params: 321
Non-trainable params: 0
```

The hidden layers have ReLU activation and the model is trained with a simple loss function based on least-square error. The corresponding flatbuffer size (*model.h*) is ≈

3KB. This model was implemented on the STM32 boards (Disco Cortex-M7), ARM Cortex-M3 soft IP and the MicroBlaze, with the corresponding run-time obtained as follows -

	Disco (Cortex-M7)	**ARM IP (Cortex-M3)**	**MicroBlaze**
Clock freq. (MHz)	100	50	100
Inference Time (μs)	**140**	**512**	**113**

Table 6.1: Inference timings for Sine model (Default kernels)

Note that here the inference engine uses the default floating-precision kernels.

DSP Acceleration in MNIST Model:

This model performs the classification of numbers in the MNIST dataset (28x28 images) with a test accuracy of **97.13%**. This model has **50890** parameters and 2 fully connected layer with the following architecture -

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 64)                50240
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
```

The hidden layer has ReLU activation and the output layer has Softmax activation. This model is implemented on the STM32 boards (Disco Cortex-M7) and the fully connected kernel is also accelerated with the DSP instructions. Following are the results obtained -

	w/o DSP	**w DSP**
Inference Time (μs)	**11238**	**9332**

Table 6.2: Inference timings for MNIST model on Disco (Cortex-M7) board

The corresponding flatbuffer (*model.h*) size is $\approx$ **200KB**. This model is not implemented on the ARM soft-IP or MicroBlaze due to the limited memory available on the CMOD A7-35T board.

6.2 Integer Models

The above MNIST model is quantized so that all the model parameters can be expressed using 8-bit signed integers. As a result, the corresponding flatbuffer (*model.h*) size reduces to $\approx$ **52KB**. The model accuracy on the test data is **97.16%**.
This model is implemented on the STM32 boards (Disco Cortex-M7) and MicroBlaze, with the corresponding run-time using integer default kernels obtained as follows -

	Disco (Cortex-M7)	**MicroBlaze**
Inference Time (μs)	**15777**	**8674**

Table 6.3: Inference timings for Quantized MNIST model (Default kernels)

Again, this model is not implemented on ARM soft-IP due to shortage of memory. Further, the fully connected kernels are optimized using CMSIS-NN library in Disco Cortex-M7 board, and by using Custom IPs in MicroBlaze. The corresponding run-time is reported as follows -

	Disco (Cortex-M7)	**MicroBlaze**
Optimization	CMSIS-NN	Custom IP
Inference Time (μs)	**2390**	**5007**

Table 6.4: Inference timings for Quantized MNIST model (Optimized kernels)

As evident, the CMSIS-NN kernel available for Cortex-M cores is faster than the other alternatives, which on the other hand, are much more flexible and cheaper.

6.3 Resource Utilization

Both the ARM Cortex-M3 soft IP and MicroBlaze are implemented on the Artix-7 FPGA. The corresponding resource utilization for these processors is reported as follows -

	LUT	**Flip-Flop**	**BRAM_36K**	**DSP**
Cortex-M3 IP	13261	4492	48	3
MicroBlaze	3082	2446	47	5

Table 6.5: FPGA Resource utilization by soft processors

The Memory footprint of the final generated .elf file is also an important indicator of the memory consumption. These are reported as follows -

	text	data	bss	dec
Disco (Cortex-M7)	43768	3484	4248	51500
MicroBlaze	171784	4480	6752	183016

Table 6.6: Memory footprint for Sine model implementation

	text	data	bss	dec
Disco (Cortex-M7)	55896	53568	53504	162968
MicroBlaze	195232	54576	15024	264832

Table 6.7: Memory footprint for MNIST model implementation (integer)

As evident, MicroBlaze code consumes a lot more memory as compared to the hardcore Cortex-M7 counterpart. This is also one of the cons for using microblaze. This difference can be attributed to the fact that these processors follow different ISA. The compiler also plays a definite role in reducing the code memory (MicroBlaze compiler vs GCC ARM toolchain). These design constraints and performance trade-offs must be considered while selecting an appropriate hardware.

The previous discussions were related to the toolchain development. In the upcoming chapters, we will apply the discussed methodologies to work on a Genomics-based application of DNA pairwise sequence alignment using the technique of Deep Reinforcement learning. This would allow us to demonstrate a powerful machine learning-based application on the edge devices.

Chapter 7

Pairwise DNA Sequence Alignment on Edge Devices

The previous chapters were focused on the tool-chain development, essential for deploying different kinds of TinyML applications on the edge-devices. In this chapter, we will discuss about one such application which has been implemented and deployed on the edge-devices using the previously discussed methodology. We will discuss about various design techniques and implementation challenges while discussing about the Pairwise DNA sequence alignment application.

7.1 Introduction

DNA Sequence Alignment is one of the prime applications in the field of Computational Genomics. Sequence alignment results are used to compare two or more sequences with unknown functions in the order of base arrangement, to other sequences with known functions. Each nucleotide DNA sequence is composed of 4 bases: Adenine (A), Cytosine (C), Guanine (G) and Thymine (T). DNA alignment reflects the similarity between these sequences and their biology features. There are 2 types of sequence alignments [11] -

Pairwise Sequence Alignment, which is used to identify regions of similarity that may indicate functional, structural and/or evolutionary relationships between **two** biological sequences (protein or nucleic acid).

Multiple Sequence Alignment (MSA) is the alignment of **three or more** biological sequences of similar length. From the output of MSA applications, homology can be inferred and the evolutionary relationship between the sequences studied.

Rest of the discussion is based on performing the pairwise sequence alignment.

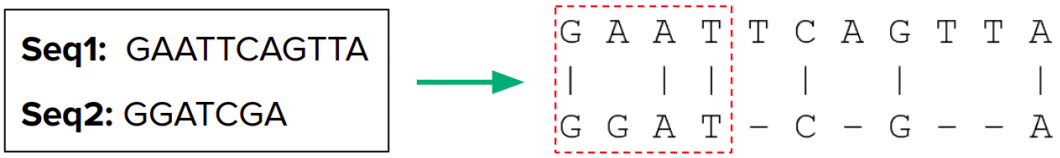


Figure 7.1: Example - Pairwise sequence alignment. Total number of matches and mis-matches in the above alignment are 6 and 1, respectively. Few gaps (4) are inserted in the later sequence.

The above figure shows an example of the pairwise sequence alignment. A major objective is to perform the alignment in such a way that we obtain the **maximum** number of matches among the 2 sequences. However, a more qualitative comparison of different alignments produced for the same set of sequences can be carried out using an Alignment score (discussed later).
The pairwise sequence alignment task is a dynamic programming problem, which is NP hard. This signifies that the number of computations required would grow exponentially with the length of the input sequences. This can prove to be a bottleneck while running the sequence alignment job with larger sequences on a resource-constrained device (limited memory, bandwidth, etc.).
Pairwise alignment is commonly performed using various traditional tools such as Banded alignment, BLAST, MUMmer [28], Clustal Omega (MSA) [25], etc. along with various cloud processing solutions. Out of these, **BLAST** [12, 13, 27] is the most commonly used tool. We can even download and execute the BLAST program on a local device such as NVIDIA Jetson-Nano kit, etc. This can be very useful while bench-marking the performance of a newly created alignment tool. BLAST is also often used as part of other algorithms that require approximate sequence matching. It's emphasis is more on the speed and for most of the cases, it cannot guarantee the optimal alignments of the query and database sequences. The scoring system of the BLAST tool would also form the basis for calculating our alignment scores.

Thus, our **problem statement** is to design and deploy a pairwise DNA sequence alignment tool which performs an end-to-end alignment of the input sequences. Also, we would be using the Deep Reinforcement Learning (**DeepRL**) approach for tackling this problem. The pros and cons associated with this method would be highlighted later. However, this allows us to implement and deploy a powerful TinyML application on an edge-device using the previously discussed methodologies.

7.2 Why deploy on Edge devices?

Sequence alignment is a way of arranging the obtained sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. They are useful in bio-informatics for identifying sequence similarity, producing phylogenetic trees, developing homology models of protein structures and is the most frequently used method for comparative analysis of biological genomes.

At least half of the world's population cannot obtain essential health services. Genomic medicine has the potential to make genetic diagnosis of disease a more efficient and cost-effective process, by reducing genetic testing to a single analysis, which then informs individuals throughout life [32]. Genetics will be important not only to understanding the cause of a disease, but also to recognizing the manner in which an individual responds to particular therapies.

Compared to developed nations, developing countries have many healthcare issues, making them more vulnerable to the harmful consequence of infectious and non-communicable diseases. The mortality rates from infectious diseases like Malaria, AIDS, Tuberculosis are typically higher in developing countries owing to the poor healthcare infrastructure. Point-of-care testing (**POCT**) [33] is a rapidly growing diagnostic tool that has improved delayed testing challenges in resource-limited settings worldwide, especially in areas with the unavailability of modern laboratory equipment and trained human resources. With such a high disease burden and strong demand, it leads to an urgent need for the research and development of newer, advanced, reliable and easy-to-implement POCT methods and devices which should also be of low cost and maintenance.

Nanopore sequencing [34] is a unique, scalable technology that enables direct, real-time analysis of long DNA or RNA fragments. **MinION** [35] is one such portable nanopore sequencing device that can be easily operated in the field with features including monitoring of run progress and selective sequencing. *Integrating DNA sequencing along with real-time sequence alignment on the edge-devices would be highly beneficial for diagnosis in remote areas with resource-constrained settings. Such a system can be operated with basic human training without the need of any cloud servers. Availability and implementation of such technology may quickly diminish the disease impact in resource-limited regions and reduce the overall public health burden, especially in laboratory-free settings.

7.3 Deep Reinforcement Learning

Deep Reinforcement learning is a very popular topic in the machine learning domain. This method has gained immense attraction in the recent years due to its vast applications and astonishing results obtained in the field of robotics, autonomous driving and navigation, human-like behaviour, system modelling, etc. The ability of an agent to get trained within an environment (either real or simulated) is a boon for applications such as autonomous driving, etc. whereby it's very difficult to gather a structured labelled dataset. As we will observe, this can be used for solving various dynamic programming problems such as sequence alignment. In this section, we will be discussing about Reinforcement learning and how Deep NN can be used further to reduce the complexity of the problem. Further, we will discuss about various variables in the system such as the reward system, policies, etc.

7.3.1 Reinforcement Learning

Reinforcement learning (RL) is a process in which an agent learns to make decisions through trial and error. These problems can often be modeled mathematically as a Markov decision process (MDP), where an agent at every timestep is in a state **s**, takes action **a**, receives a scalar reward and transitions to the next state **s'**. During this repetitive process, the agent attempts to learn an optimum policy $\pi(a|s)$, or a map from possible states/observations to actions, in order to maximize its rewards. We either reward the agent for the desired behaviors, or we punish it for the undesired ones. The goal is to maximize the obtained rewards.

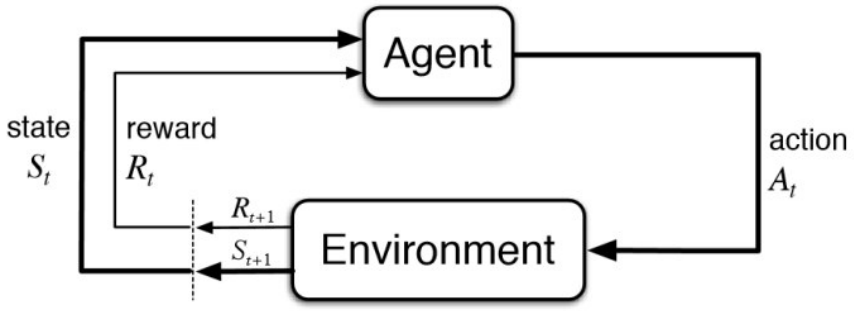


Figure 7.2: Agent observes the current environment/state and picks a suitable action for maximizing the obtained rewards, which in-turn updates the current environment. This continuous feedback allows the agent to learn an optimal policy.

At this point, it's important to highlight few important terminologies that we often use in the RL domain -

- **Environment**: Physical or simulated world in which the agent operates
- **State**: Current situation of the agent in the environment
- **Action**: An action taken by the agent by observing a state
- **Reward**: Feedback from the environment
- **Policy**: Method to map agent's state to actions
- **Value**: Future reward that an agent would receive by taking an action in a particular state

Most of the reinforcement learning algorithms which involve value-based agents are centered around the popular **Bellman equation**. It decomposes the value function for a state into two parts, the immediate reward plus the discounted future values. This equation simplifies the computation of the value function, such that rather than summing over multiple time steps, we can find the optimal solution of a complex problem by breaking it down into simpler, recursive sub-problems and finding their optimal solutions. The bellman equation is given as follows -

$$V(s) = max_a(R(s,a) + \gamma V(s')) \tag{7.1}$$

, where R(s,a) represents the immediate reward for action a and state s. V(s) and V(s') represents the value for current state and successor state respectively, and γ represents the discount factor. Once the values of the states have been determined, the agent can simply transverse along the direction yielding maximum values. In another version of the same equation, we can further combine the state-action pair to obtain a **Q-value**. Q(S,A) is an estimation of how good is it to take the action A at the state S. Thus, a higher Q-value signifies that it is advantageous to take that particular action at that particular state.

$$Q(s,a) = R(s,a) + \gamma max_{a'} Q(s',a') \tag{7.2}$$

A very popular strategy in RL is the **Q-learning** approach, where we create what's called a Q-table or a matrix. For every episode or sequence of finite events, we fill in this table for every pair of possible states and actions. This Q-table becomes a

reference table for our agent to select the best action based on the Q-values. This table is updated iteratively for all the trials. These updates occur after each step or action and ends when an episode is done. Done can simply mean reaching some terminal point by the agent such as arriving at a destination, completion of sequence alignment, etc.

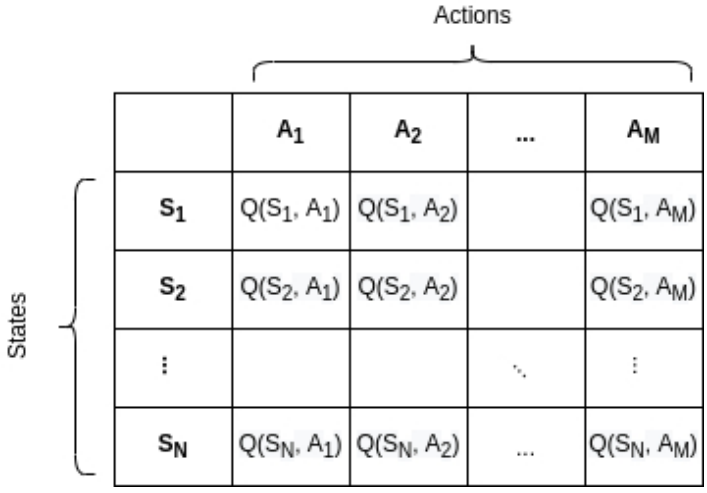


Figure 7.3: Q-table

An agent interacts with the environment in two ways. The first is to use the Q-table as a reference and view all possible actions for a given state. The agent then selects the action based on the max Q-value of those actions. This is known as **exploiting** since we use the information we have available to us to make a decision. The second way to take action is to act randomly. This is called **exploring**. Instead of selecting actions based on the max future reward we select an action at random. Acting randomly is important because it allows the agent to explore and discover new states that otherwise may not be selected during the exploitation process. During the training, it is important to create a balance between the extent of exploration and exploitation.

*The above Q-table method seems as a reliable solution for a smaller problem with fewer states. However, in many practical decision making problems, the states are high-dimensional (eg. images from a camera or the raw sensor stream from a robot) and cannot be solved by traditional RL algorithms. As an example, the number of states in the *Go game* is even more than the number of atoms in the known universe. Thus, it's very difficult to construct a Q-table corresponding to these complex systems. Next, we will talk about how Deep NNs could help us in such situation.

7.3.2 Deep Reinforcement Learning

Deep neural networks are known to be excellent function approximators. They are used to build state-of-art systems in various applications such as image recognition, speech recognition, natural language process and others. The result that neural networks are universal approximators is one of the theoretical results most frequently cited to justify the use of neural networks in these applications [14].

As an approximation, we can replace the Q-table with a Neural network. Rather than mapping a state-action pair to a Q-value, a neural network maps input states to (action, Q-value) pairs. In simple words, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. This enables us to even handle some high-dimensional data such as images, etc. which was previously challenging.

$$Q(s,a) \approx Q'(s,a,\theta) \quad [\theta : NN\ hyperparameters] \tag{7.3}$$

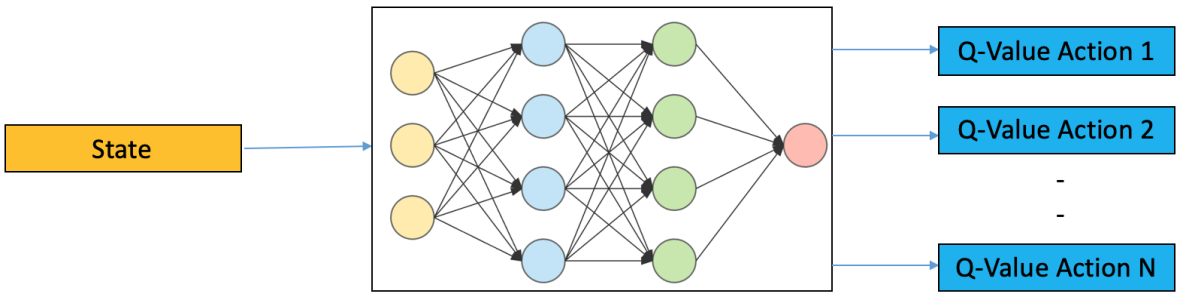


Figure 7.4: Deep Q-Learning

For training a Deep Q-network, the loss function is typically the mean squared error of the predicted Q-value (Q') and the target Q-value (bellman equation).

$$loss = (R(s,a) + \gamma max_{a'} Q(s',a') - Q'(s,a))^2 \tag{7.4}$$

During the training, the agent also stores its experiences and uses it later. This is also known as Experience Replay. Initially, we do not know the target or actual Q-values here as we are dealing with a reinforcement learning problem. However, we can argue that it is predicting its own value, but since R is the unbiased true reward, the network is going to update its gradient using backpropagation to finally **converge**. The convergence of the Deep RL problems is often challenging and it depends on a lot of factors including learning rate, exploration-exploitation balance, reward system, policies, etc. Next, we will discuss about the role played by Rewards and Policies during the training of Deep RL networks.

7.3.3 Effect of Policy and Rewards

Rewards highlight the feedback received from the environment about how good the action was in a particular state. Mathematically, rewards can either be positive (enforcing desired actions), or negative (avoiding undesired actions). On the other hand, policy underlines the methodology via which an agent samples a particular action in a given state. The reward system and the action policy play a very crucial role in determining the accuracy and convergence of the RL systems. To understand this in detail, let us consider an example of the *Coast Runners* game.

In Coast Runners game, different motorboats race against each other to reach the finish line. Whoever reaches the finish line first, wins the game. Coast runners does not directly reward the player's progression around the course, instead the player earns higher scores by hitting targets laid out along the route. All along the track, there are various speed boosters (highlighted with blue sparks). If a motorboat comes in contact with these speed boosters, it's speed will increase and it would also acquire few additional game points. Now, consider a scenario whereby we are teaching a Deep RL agent to learn and play this game. The environment or state at any timestamp could be the image snapshot of the game's interactive window. The actions could be to either row left, right, or do nothing.

Figure 7.5: Coast Runners game

Let us assume that the score a player earns would reflect the informal goal of finishing the race, so we included the game score in an internal benchmark designed to measure the performance of RL systems on racing games. However, it turns out that the targets or even the speed boosters were laid out in such a way that the RL

agent could gain a high score *without having to finish the track*. As a result, we might observe some unexpected behavior while training the RL agent to play the game. For ex, the motorboat might spiral around and hit the same object or booster as it is still gaining the game points, instead of ever completing the race.

The previous example highlighted what happens when a misspecified reward function encourages an RL agent to subvert its environment by prioritizing the acquisition of reward signals above other measures of success. Instead, a better reward system in a race could have been observing the game time, etc. as a vast majority of humans would seek to complete the racecourse. Thus, carefully designing a reward system is very crucial for the overall stability and accuracy of the RL agent.

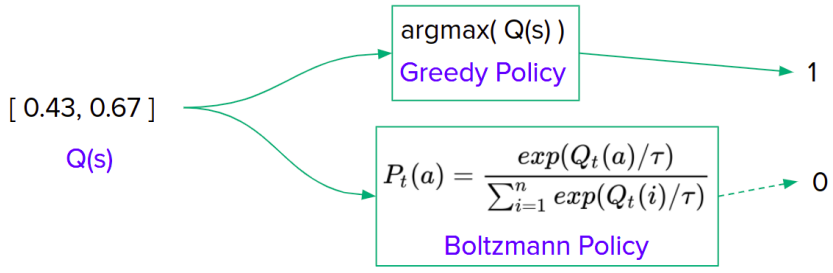
In deep Q-learning, we obtain the Q-values for each action at the output of the network. A policy function specifies how to sample a particular action given these Q-values. As an example, we might become greedy and select the action yielding the highest Q-value. This is precisely known as a **Greedy** policy.

$$action = argmax_a Q(s,a) \quad [Greedy\ policy] \tag{7.5}$$

On the other hand, we might want to sample an action from a probability distribution, which is inherently computed using the obtained Q-values. **Boltzmann** policy is one such policy. Boltzmann policy turns the agent's exploration behavior into a spectrum between picking the action randomly (random policy) and always picking the most optimal action (greedy policy). It is a probabilistic approach for choosing an action. The following equation highlights the probability for selecting action a -

$$p(a) = \frac{exp(Q(s,a)/\tau)}{\sum_{i=1}^{n} exp(Q(s,i)/\tau)} \tag{7.6}$$

The choice of a policy often depends on the type of the application. A greedy policy could be more suitable for deterministic systems such as sequence alignment, etc. where the choice of an action is influenced by a fixed pattern in the input data (discussed later). Different policies can output different actions for the same inputs. Next, we will discuss about the Dueling Network Architectures for Deep RL.



7.3.4 Dueling Deep Q-Networks

Most of the approaches for deep RL use standard neural networks, such as convolutional networks, MLPs, LSTMs and autoencoders. However, it would be beneficial if we could generalize learning across different actions without imposing any change to the underlying reinforcement learning algorithm. This allows us to encourage only those actions which are advantageous.

One such network is know as Dueling Deep Q-network. Wang et al. [15] presents the novel dueling architecture which explicitly separates the representation of state values and state-dependent action advantages via two separate streams. The key motivation behind this architecture is that it is unnecessary to know the value of each action at every timestep, rather than knowing which action is more advantageous. The dueling architecture consists of two streams that represent the value and advantage functions, while sharing a common convolutional feature learning module. The two streams are combined via a special aggregating layer to produce an estimate of the state-action value function *Q(s,a)*. In the experiments conducted by Wang et al., the value-advantage function is aggregated as follows to represent the Q-value function-

$$Q(s,a,\theta,\theta') = V(s,\theta) + \left(A(s,a,\theta') - \frac{1}{|A|} \sum_{a'} A(s,a',\theta') \right) \tag{7.7}$$

, where θ and θ' represent the model parameters for the value and advantage streams, respectively. V and A represent the value and advantage functions, respectively. Since the dueling architecture shares the same input-output interface with the standard DQN architecture, the training process is identical. The mean term in the aggregation function also acts as a L1-regularizer during the training process.

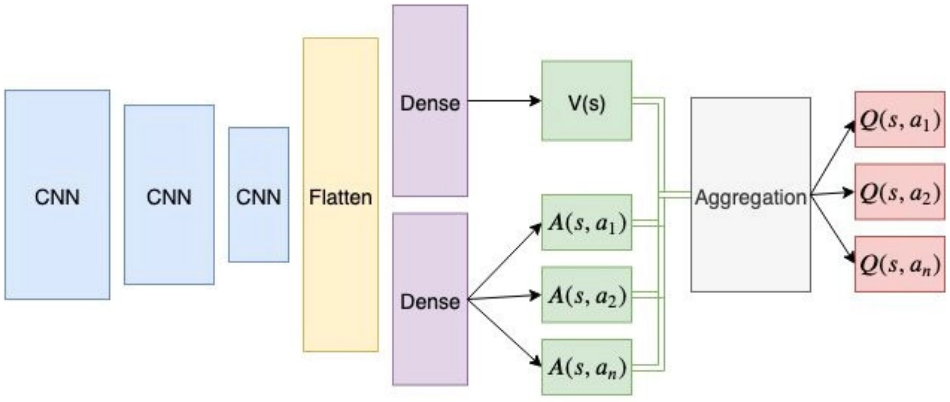


Figure 7.6: Dueling Deep Q-Network Architecture

7.4 DNA Sequence Alignment using DeepRL

In order to use deep RL for the sequence alignment problem, we must first specify the various system parameters including the environment, actions, reward system, etc. Most of the discussion in this section is referred from the deep RL paper by Song et al. [18] on the Pairwise heuristic sequence alignment algorithm. Let us discuss each of them individually.

7.4.1 Environment

The environment for the deep RL problem is defined by a **finite window** consisting of nucleotide pairs from both the sequences. Each of the type of nucleotide (A,C,G,T) was converted to a **3x3** pixel square with **CMYK** color representation. To separate the left, right, top, and bottom end of the sub-sequences, 3x3 pixels of empty space are added. Thus, the DNA information is converted into an image representation. This window image is the input state for the deep Q-network at some timestamp. The network then predicts a suitable action based on the current state, which is also further used to update the window or the environment.

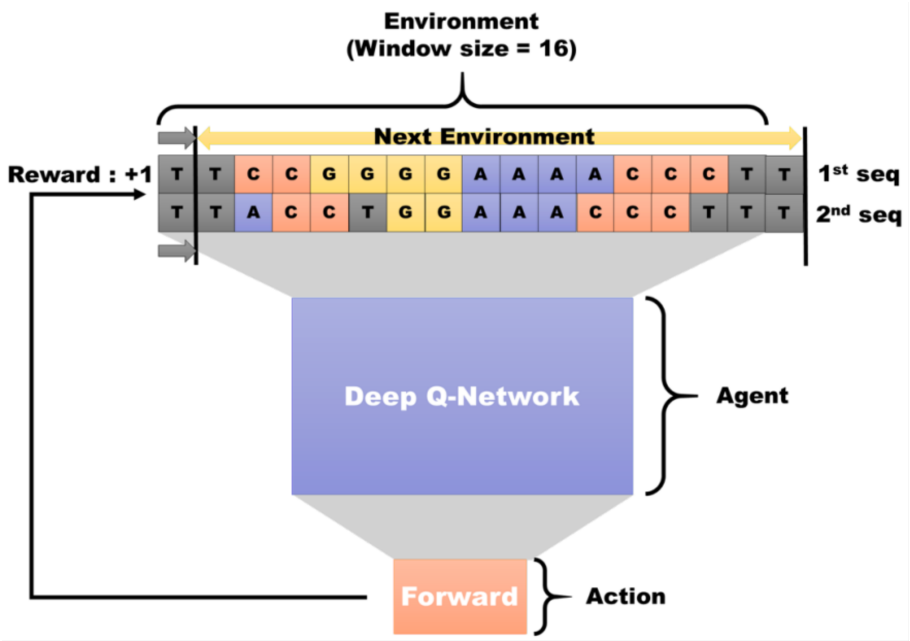


Figure 7.7: RL defines the sub-sequences in current window as environment. The overall sequence alignment is performed by repeating the small alignments (local best path selection method).

As an example, if the action is forward, the window simply shifts to the right by 1-position for both the sequences. More details regarding the actions is discussed later. The window formation starts from the beginning of both the sequences, and it continues till it reaches the end of either of the sequences. The following flow chart highlights the update rule for the environment -

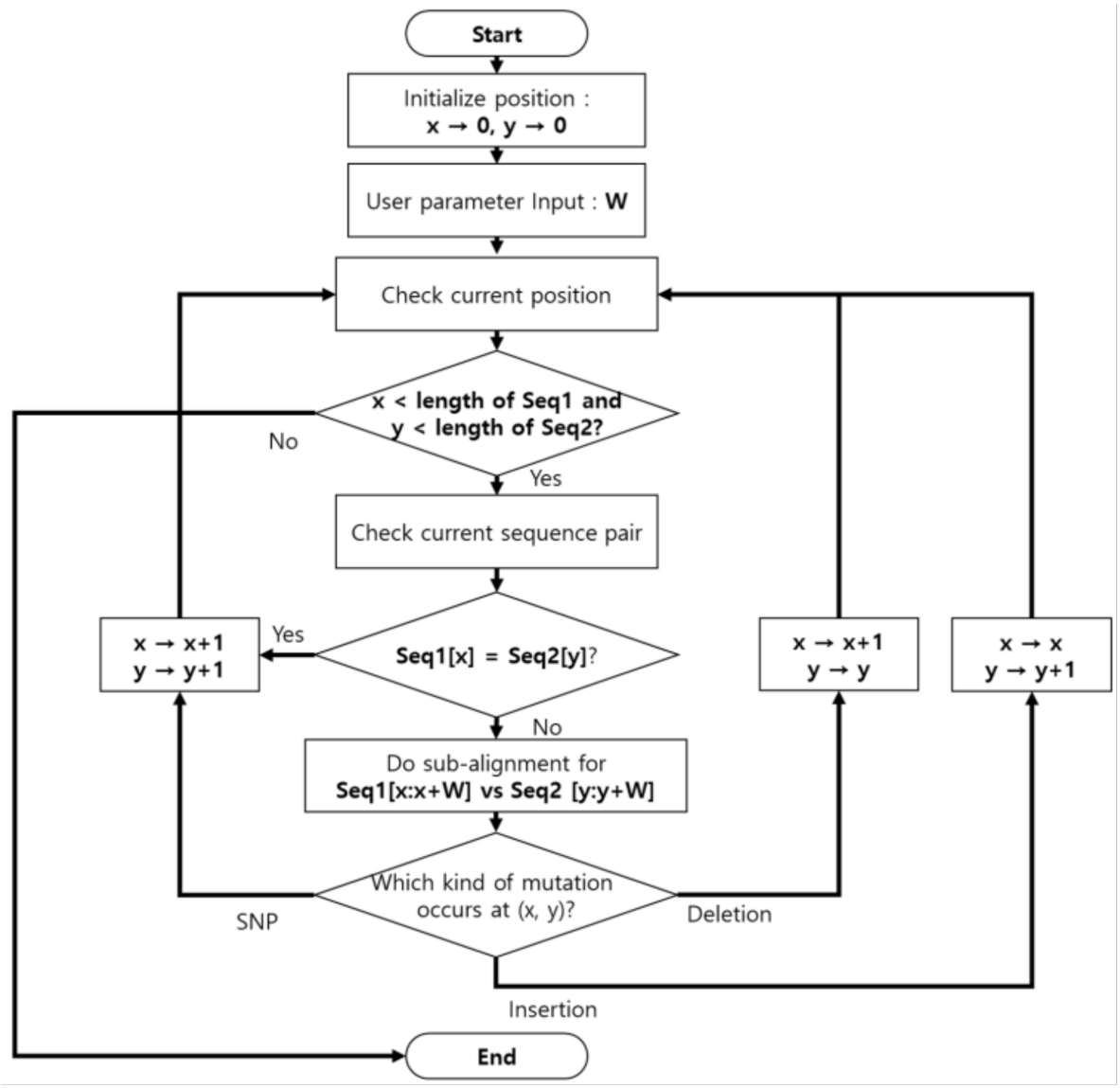


Figure 7.8: Update rule for the window environment. **W** represents the window size.

*Since the window size is finite, the number of computations or required resources is **fixed** for a sub-alignment. This is highly advantageous for larger sequences as they can still be handled using finite resources.

7.4.2 Actions

There are 3 possible discrete actions in the pairwise sequence alignment problem: **Forward**, **Insertion** and **Deletion**.

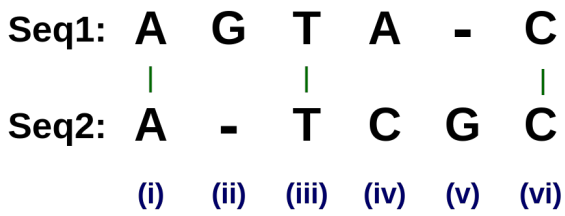


Figure 7.9: Possible actions in the pairwise DNA sequence alignment. (i), (iii), (iv) and (vi) correspond to the forward action. (ii) and (v) correspond to the insertion and deletion actions, respectively.

In case of **forward** action, the agent simply shifts the window to the right by 1-position for both the sequences. This is synonyms to the agent placing a match at the current index. The ultimate result could be either a true nucleotide match (i, iii) or a mis-match (iv), however, the agent simply decided to go forward.
In case of **insertion**, the agent inserts a gap in the 2^{nd} sequence (ii). It also shifts the window to the right by 1-position for the 1^{st} sequence. An insertion can be thought of an extra nucleotide that was inserted in the 1^{st} sequence. The corresponding gap is inserted to balance this effect.
In case of **deletion**, the agent inserts a gap in the 1^{st} sequence (v). It also shifts the window to the right by 1-position for the 2^{nd} sequence. A deletion can be thought of a nucleotide that went missing in the 1^{st} sequence. The corresponding gap is inserted to balance this effect.

The concept of insertion and deletion is associated with respect to the 1^{st} sequence. The opposite is true for the other sequence. Also, forward action is often abbreviated as SNP or single nucleotide polymorphism. This can either mean a match or a mis-match at the current index. These 3 actions will correspond to 3 different Q-values at the output of the deep Q-network.

Based on these actions, the window is updated after every iteration. As we discussed earlier, the space complexity of this approach is constant (**O(1)**) due to the fixed window and network size. The time-complexity is linear (**O(n)**) as it scales linearly with the number of iterations. Next, we will discuss about the scoring mechanism for the generated alignment and the reward system.

7.4.3 Alignment Score and Reward System

The sequence alignment tool aligns the given 2 sequences in such a way that we obtain the maximum number of matches. However, an alignment with too many gaps is also not desirable. Thus, it becomes important to characterize the alignment results and find a suitable metric for analysing the quality of the generated alignment.

One such metric is the BLAST alignment score. It uses a rule which assigns a numerical score to any alignment; the higher the score, the better the alignment. This alignment score is computed by assigning a value to each aligned pair of letters or nucleotides, and then summing these values over the length of the alignment [30]. For nucleotide alignments, the default BLAST options uses a reward of (+1) for a nucleic match, penalty of (-3) for each mis-match. The creation or opening of a gap in an alignment results in a negative "gap-opening" penalty (-5), with each extension of a pre-existing gap incurring a lesser penalty (-2).

Match	Mis-Match	Gap Opening	Gap Extension
+1	-3	-5	-2

Table 7.1: BLAST default scoring for nucleotide alignment [29]

Consider an example of the following alignment for the calculation of the alignment score. This alignment produces 2 matches and 1 mis-match. There is a single gap-opening, which is further extended twice.

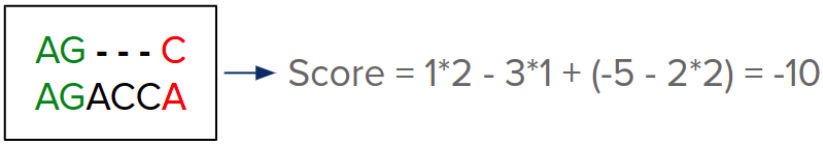


Figure 7.10: Alignment score calculation

Thus, given various alignment results, we can assign a numerical value corresponding to their quality of alignment. This helps in differentiating bad results from the good alignment results. This will be further used to benchmark our results.

The BLAST alignment score should ideally also influence the reward system used since the obtained alignment results from the RL tool would be benchmarked using this scoring mechanism. This is important as the reward system forms the basis of the training process. Let's discuss this in detail.

The rewards for mis-match, gap opening and gap extension is same as the values used in the BLAST alignment score. Further, we normalize these values in the range of (-1, 1). On the other hand, the reward for a match is chosen to be (+1) in order to create a bias towards getting more number of matches. As we will further observe, this bias results in a stable and accurate training. The following table summarizes the reward system used for training the deep RL agent -

Match	Mis-Match	Gap Opening	Gap Extension
+1	-0.6	-1	-0.4

Table 7.2: Reward system used for training the deep RL agent

7.4.4 Policy and Network Architecture

The ordering or pattern of the nucleotides in a given window of sub-sequences decides which action might be the most beneficial. Since the action can be determined by observing this pattern, we can consider it as a deterministic system. We have chosen the **Greedy** policy for determining the action from the obtained Q-values at the output of the network.

$$\Pi(s,a) = argmax_a Q(s,a) = \begin{cases} 0, & \text{Forward} \\ 1, & \text{Insertion} \\ 2, & \text{Deletion} \end{cases}$$

We have used a CNN architecture for modelling the DQN agent, which is similar to what is presented in the work by Song et al. [18]. In a way, this problem is an image classification problem, the difference being the way in which the model gets trained. CNN is the most widely used network architecture for image classification tasks due to its dimensionality reduction, feature sharing, and pattern finding.

We have also used a dueling network architecture which distributes the predicted reward (Q value) into a kind of average and variance, and each of them is called as "Value" and "Advantage", respectively [7.3.4]. Using this method, the agent can learn the scores of the states(value) and actions separately, which helps in improving the stability and convergence of the learning process. Thus, the last layer of the model has 4 output neurons corresponding to the state value and the 3 different actions. Since these outputs are continuous in space, the corresponding activation

function is linear. Following figure shows the detailed network architecture and the model parameters.

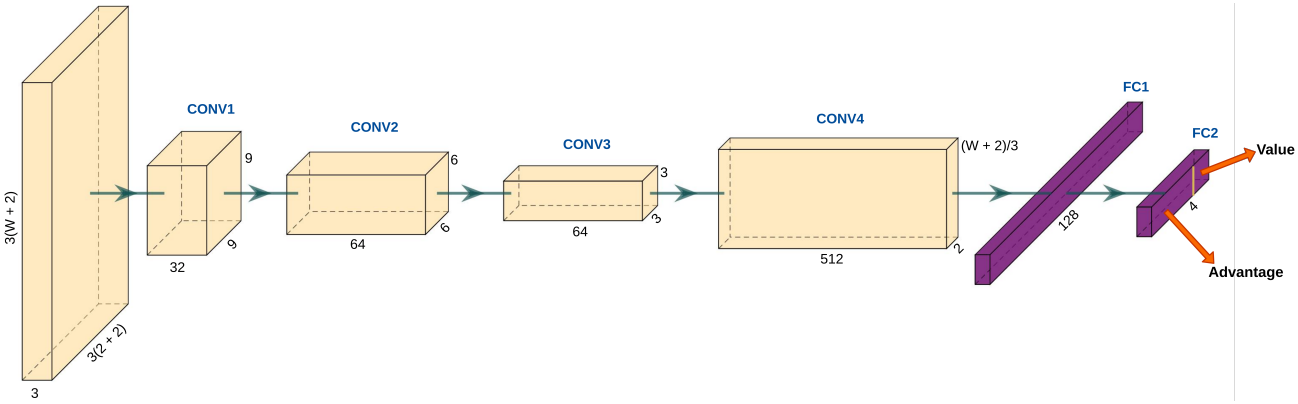


Figure 7.11: Network architecture for Dueling Deep Q-network

	Input size	Filter shape	Stride	Padding
Conv1	3(W+2) x 12 x 3	9 x 9 x 32	3 x 3	Same
Conv2	(W+2) x 4 x 32	6 x 6 x 64	3 x 3	Same
Conv3	[(W+2)/3] x 2 x 64	3 x 3 x 64	1 x 1	Same
Conv4	[(W+2)/3] x 2 x 64	[(W+2)/3] x 2 x 512	1 x 1	Valid
FC1	512	128	-	-
FC2	128	4	-	-

Table 7.3: Parameters of the Network (W = Window size)

The value and advantage outputs are further aggregated to give the Q-values, as discussed earlier. Currently, a single fully-connected layer (FC2) is computing both the value and advantage outputs. We could also design different networks for handling the value and advantage output streams separately. This might lead to higher accuracy, however, this would further increase the model parameters, resource consumption and inference timing. As we will further observe, the current network has too many parameters for being able to get deployed on a resource-constrained device. We shall reduce the model parameters without much compromising with the accuracy. A lot of designs were tried out, however, we have achieved the best accuracy so far using the above network architecture.

7.4.5 Where to Start?

The RL approach is a sequential method to deal with the problem of sequence alignment. Once a decision or action has been taken for the sub-sequences in a window, we can't modify it later as the window keeps moving forward. Thus, the effect of the starting point on the result is critical because the alignment procedure is performed only towards one direction. We can remove such instability by applying the preprocessing methods of the conventional alignment methods such as longest-common substring, the Clustal Omega and the MUMmer [18].

We have used the longest-common substring method in our implementation. Given the two input sequences, we first find the longest substring which is common to both of them. Using this longest-common substring (**LCS**) as the starting point, we further split the alignment task into 2 sub-alignment jobs. In one case (for all sub-sequences to the left of LCS), we perform the alignment in the backward direction. On the other hand, for all sub-sequences to the right of LCS, we perform the alignment in the forward direction. The aligned sub-sequences output from these 2 different sub-alignment jobs is further stitched back to the LCS in a similar fashion. The following figure summarizes the LCS method -

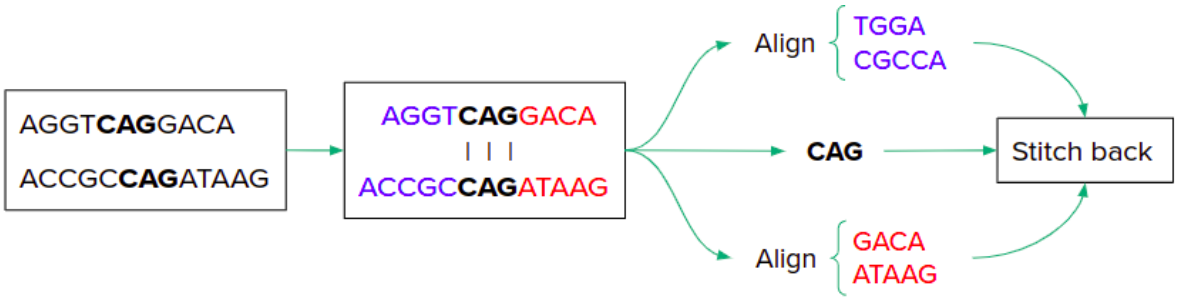


Figure 7.12: Longest-common substring as the starting point in sequence alignment

Other pre-processing methods can also be tried out. We proceeded with the LCS approach as it is easier to implement on Python. However, the time complexity of the LCS algorithm for 2 sequences of length m and n, respectively is O(mn). Thus, it might become infeasible to use this method with very large sequences (human genomes having length of 4 million etc.). In that case, we can eliminate this method and only proceed with the forward alignment.

We have discussed all the components required for building the RL agent. The following figure shows the complete end-to-end flow of the pairwise sequence alingment using DeepRL.

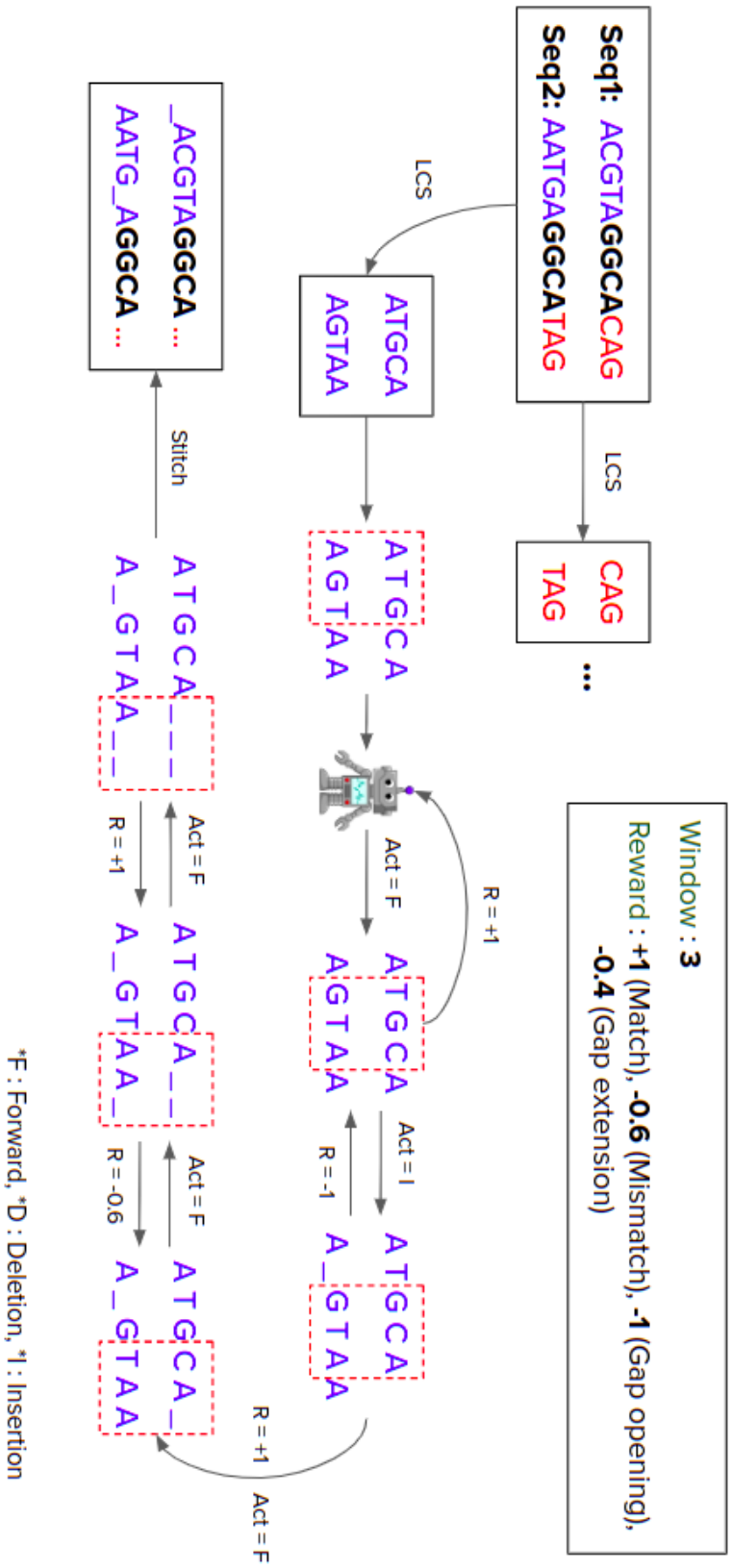


Figure 7.13: Pairwise sequence alignment using Deep RL - Complete flow

7.5 Dataset

Every deep leaning model requires a dataset for training, whether labelled or unlabelled. For the pairwise sequence alignment problem, we don't have a definite labelled dataset, i.e actions corresponding to every window of sub-sequences. Instead, we would like to simulate the sequence alignment environment using the artificially generated random sequences. These sequences must be generated using a definite rule and shall be able to mimic various mutations such as SNP (single nucleotide polymorphism), Insertion and Deletion (InDels).

First, we generated a completely **random** sequence. Then, we generated the other sequence by mutating this random sequence. For convenience, we used the **JC69 model** to create the SNP mutations [31]. Similarly, we used the **Zipfian distribution** based InDel length model for generating the InDels [36]. Everything is implemented using python as this allows us to easily integrate the sequence generator with our RL environment. Python also offers flexibility and is suitable for faster training process as the generated sequences doesn't requires any pre-processing. Once both the sequences are generated, they are further converted into an image representation as discussed earlier. Following figure summarizes the process of sequence generation and the detailed system parameters used for training -

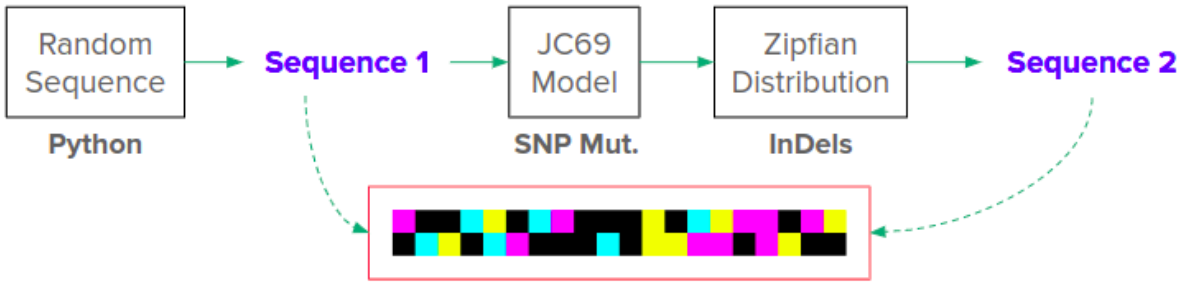


Figure 7.14: Dataset generation. Sequences are artificially generated.

Sequence length (l)	1500
Probability of SNP (p_{SNP})	0.1
Probability of indel (p_{indel})	0.05
Maximum length of indel (I_{max})	2
Zipfian distribution parameter (s)	1.6

Table 7.4: Detailed parameters of sequence generation for training procedure

7.6 Training

At this stage, we have all the required components for training our RL agent. We can start by defining our environment, agent, episode details, etc. but first, we need to choose a RL framework for training. **Keras-RL** [38] is one such open-source RL framework. It implements some state-of-the art deep reinforcement learning algorithms in Python and seamlessly integrates with the deep learning library Keras. Furthermore, Keras-RL works with OpenAI Gym (standard API for reinforcement learning, and a diverse collection of reference environments) out of the box. This results in faster implementation, easier debugging, and high flexibility in implementing a variety of algorithms. We can also extend Keras-RL according to our own needs (highly customizable). We can use built-in Keras callbacks and metrics or even define our own. Best part, it is easy to implement our own environments and even algorithms by simply extending some simple abstract classes. Many algorithms have been already implemented in Keras-RL such as Deep Q-Learning , Double DQN, Dueling network DQN, Deep SARSA, DDPG, etc. The next step would be to define our own custom environment and agent for the sequence alignment problem. All the necessary codes are available at the github repository:
https://github.com/aryanlall11/EdgeAlign.git

7.6.1 Custom Environment and Agent

We need to define the functionality of our own environment for the pairwise sequence alignment problem. The *Env* class from the *gym* library in python offers the required functions that needs to be inherited and modified for our own environment. We begin by creating an *EdgeAlignEnv* class. When we pass *Env* to this *EdgeAlignEnv* class, we inherit the methods and properties from the OpenAI Gym environment class. Following methods are required to be re-defined in our class -

- **_init_**: This is used to initialize the actions, observations, and sequences. The more general use is to define all the class variables. Since our actions are discrete, they consume a discrete space in the initialization. On the other hand, the observation space will hold an array of our current state. This belongs to the Box space as there could be many possible permutations.
- **step**: It defines what we do after we take an action. It also updates the subsequences in the alignment window according to the action taken.

- **reset**: It is used to reset our environment after each episode. Here, we can generate the new sequences for the next episode and can also display various console messages such as the total number of matches, episode reward, etc.
- **render**: It is used to visualize our results. We won't be using it in our implementation, hence we can simply skip it.

Once we have defined the above methods, our implementation of the custom environment *EdgeAlignEnv* is complete. Next, we can define our RL agent. For our implementation, we have chosen the *DQNAgent* [39] from the Keras-RL framework. The DQN agent uses the Sequential memory to store various states, actions, and rewards for experience replay. We set this storage limit as *5000*. The warm-up steps (agent under exploration) is also set at *5000*. As we will observe, the flag for the dueling network implementation is also set as *True*, with the dueling type as *Average*. We are using the the Greedy policy. Please refer to the codes for more detailed implementation. Next, we can simply infer these classes and start the training procedure.

7.6.2 Training Procedure

We start by creating the objects of our environment and agent class. We also build the neural network model as follows -

```
# Sequence alignment environment
env = EdgeAlignEnv()

# Build the model
model_obj = Model()
model = model_obj.build_model()
model.summary()          # Deep-Q network

# Reinforcement learning DQN agent
agent_obj = EdgeAlignAgent()
# Dueling is set as True and type is Average
dqn_agent = agent_obj.build_agent(model=model,
                                  enable_dueling_network=True,
                                  dueling_type='avg')
```

Lastly, we can start the training process by compiling and fitting our agent on the environment as follows. The training may take some time depending upon the window size (higher the window size, higher would be the training time), and the number of training episodes.

```
# Compile the DQN agent
dqn_agent.compile(Adam(lr = Learning_Rate), metrics = ['mae'])

# Train RL agent for Num_Episodes
dqn_agent.fit(env, nb_steps = Num_Episodes, visualize=False, verbose=1)
```

The following figure shows how the training progresses for a window size of **50**. Notice the number of matches obtained at the start of the training. It's considerably low for a sequence length of 1500, however, it increases as the training progresses. After few episodes, the number of matches nearly saturates to about $\approx$ 1300.

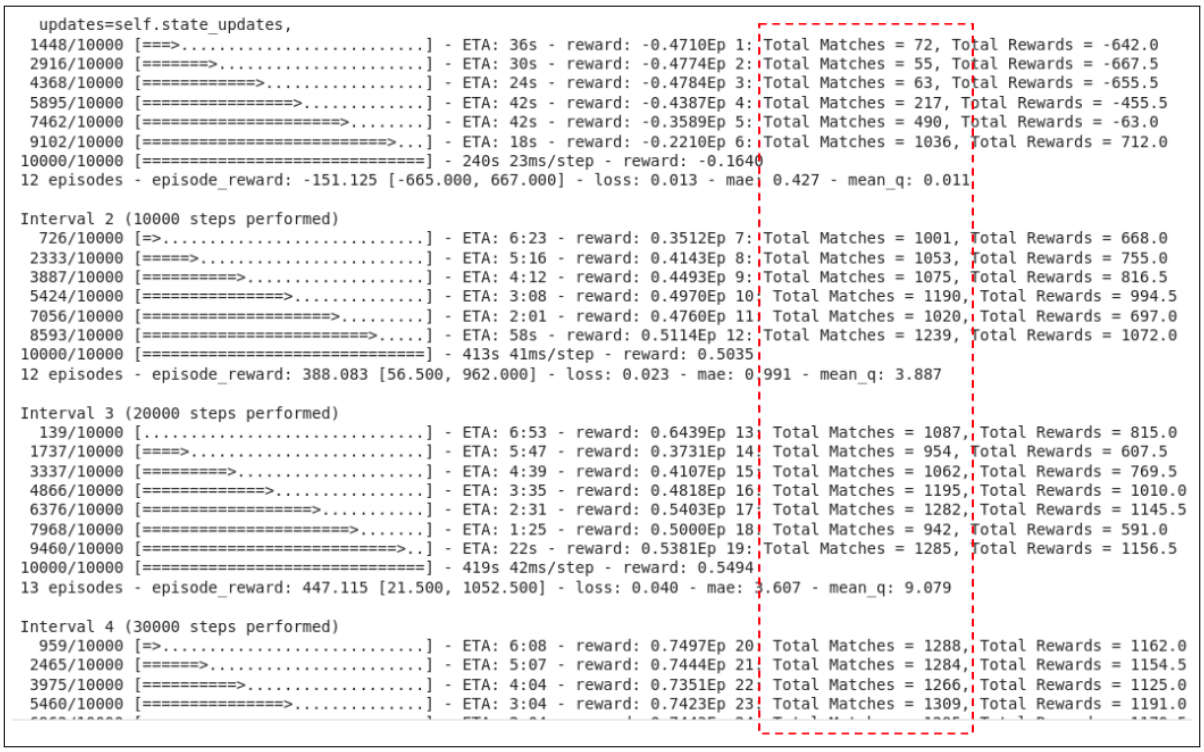


Figure 7.15: Training progress (Window: 50, Learning rate: 1e-4, Epochs: 60000)

We can try adjusting various parameters including the reward system, learning rate, etc. and observe what influence they have on the training and performance. Convergence is yet another challenge while dealing with RL agents. An improper system parameter might never lead to a converging behaviour. For example, we were not able to train the agent (didn't converge) with a learning rate of **1e-3**. The following plot shows the convergence behaviour of the RL agent for 2 different window sizes (50 & 70) during a training session. Notice how the number of matches get

saturated after few episodes. Next, we will discuss about the benchmarking results.

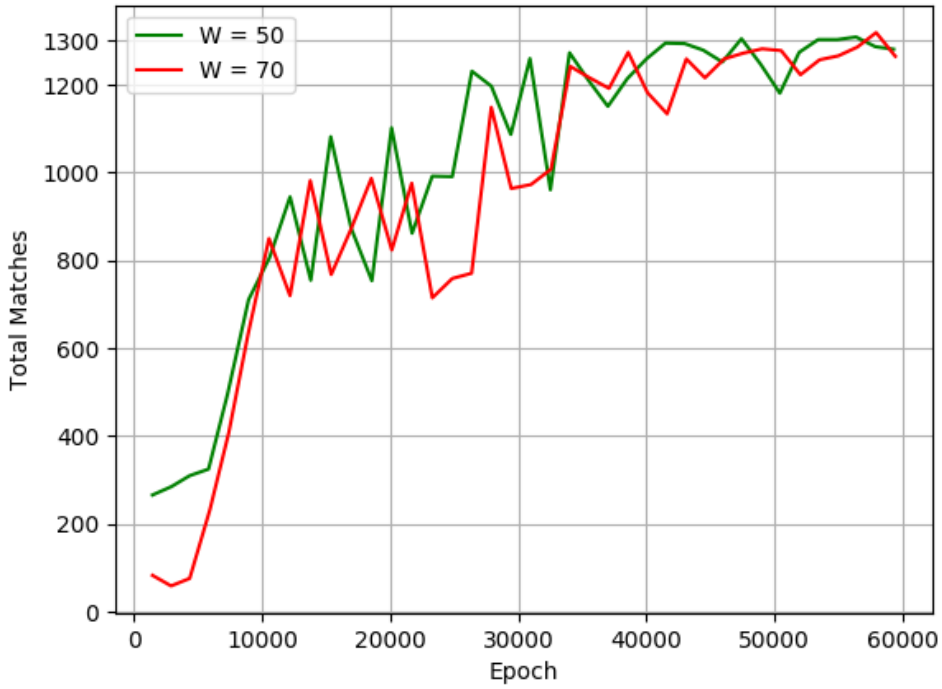


Figure 7.16: Convergence behaviour for window sizes of 50 & 70

7.7 Results

We evaluated the performance of our trained RL agent on the publicly available Influenza dataset [40]. The website also allows us to select a pair of sequences and obtain the alignment. The objective is to compare the alignment score obtained by the NCBI online tool, and our RL agent. For a fair benchmarking, we designed the following error metric for a pair of input sequences -

$$Error(E) = \begin{cases} 0, & \text{R} \geq \text{G} \\ \frac{G-R}{G}, & \text{R} < \text{G} \end{cases}$$

, where R is the alignment score obtained using the RL agent, and G is the golden alignment score obtained using the online tool. The mean error for the entire dataset (N pair of sequences) is obtained as follows -

$$Mean\ error = \frac{1}{N}\sum_{i} E_i$$

The above error metric quantifies how bad the RL model performs w.r.t to the available NCBI online alignment tool. This evaluation was conducted on a dataset of size **40**, with **5** different window sizes. Further, we deployed our RL model on the **Jetson-Nano kit** and obtained the inference timing for predicting a single action. The benchmarking results are summarized as follows -

Window size	Model parameters	Inference Timing (ms)	% Mean error
30	906116	15.668	19.19
40	1102724	16.958	16.18
50	1364868	18.361	15.01
60	1561476	21.326	13.12
70	1758084	25.774	12.47

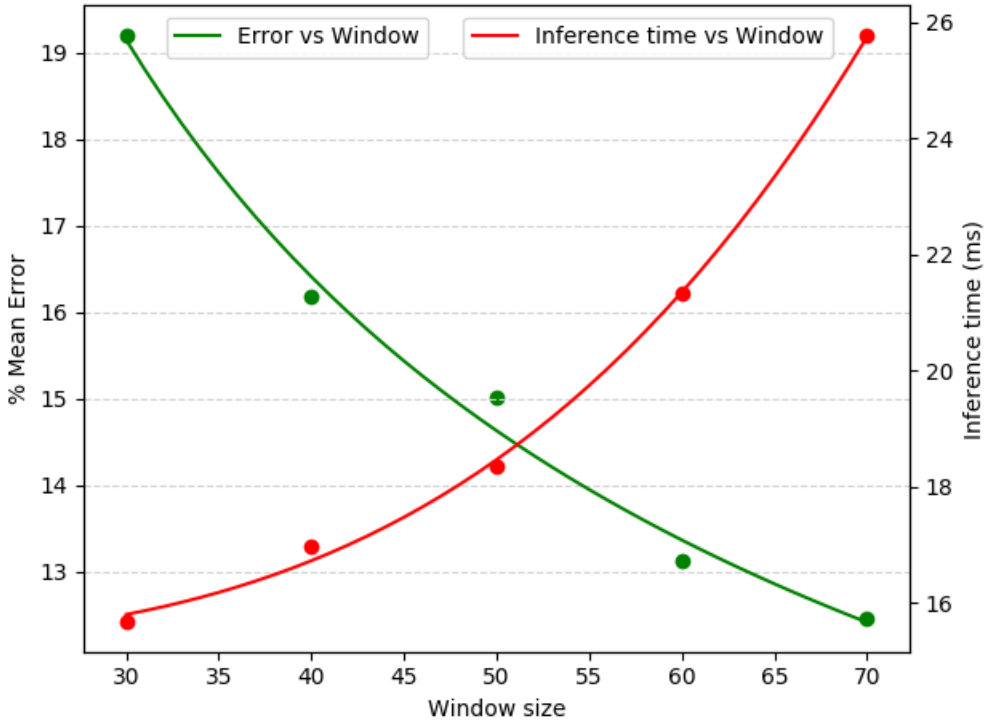


Figure 7.17: Benchmarking results on the Influenza dataset

As evident from the results, the mean error decreases with an increase in the window size. With a larger window, the agent can further observe more sequences. This increased awareness leads to better results as the chosen actions are more reliable for securing higher rewards in the long-run. However, it also increases the resource consumption and the inference timing. These trade-offs must be accounted while deciding upon the window size and the device.

7.8 Model size reduction using AutoML

The current model with a window size of say, 50 has too many parameters($\approx 13.6L$) for being able to get deployed on a resource-constrained device. This would consume a lot of memory and would also be extremely slow, considering the number of required computations. Thus, we must reduce the model size for the edge deployment. Let's discuss the different methods and how AutoML can help.

Various methods exist for compressing the model including **Quantization**, Pruning, etc. Quantization is a well known approach for compressing the model by representing the model parameters using 8-bit integers. This results in reducing the memory footprint with slight deteriotion in the model accuracy. As we will observe later, the quantization approach reduces the model size, however, it drastically deteriorates the model accuracy in the sequence alignment problem. Since the alignment is performed in a sequential and unidirectional manner, an error at some stage simply gets amplified and propagates to the later stages. Thus, quantization can't be used in problems such as sequence alignment. We need to perform all the computations in floating precision. What else can be done to reduce the model size?

7.8.1 Neural Architecture Search (NAS)

As a human, it's very difficult to try out every possible NN architecture and benchmark it for the best results. Hyper-parameter tweaking is yet another challenging and tedious task for a complex network. Questions like what should be the best network, how many CNN layers are sufficient, what should be the kernel size, number of neurons in the fully-connected layer, learning rate, etc. are quite common and often difficult to answer efficiently.

Neural architecture search (NAS) or AutoML is a technique for automating the design of Neural Network architectures. Given a labelled dataset, we can find an optimal architecture for modelling the system. Thus, we don't require to specify the network details, rather we only provide the dataset and the system itself finds an optimal architecture for training. NAS methods explore a lot of potential solutions with variable complexities and hence are very computationally expensive. The larger is their search spaces, the more there are architectures to test, train and evaluate. The search space can be restricted by various parameters such as the maximum allowed model size, operations, and latency. NAS is currently in the experimental stage, however, it is a powerful tool with huge potential.

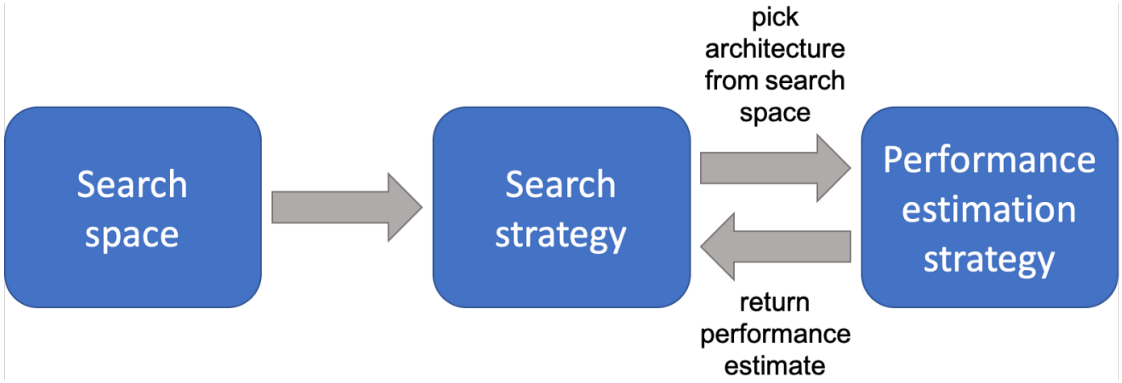


Figure 7.18: Network Architecture Search

Next, we will look at how we can implement and utilise the power of NAS using an open-source tool - AutoKeras, it's usage and how it could help us in the sequence alignment problem.

7.8.2 AutoKeras

AutoKeras is an open-source AutoML system based on the Keras framework [37]. It provides a simple and effective approach for automatically finding top-performing models for a wide range of predictive modeling tasks, including tabular or so-called structured classification and regression datasets. The user is only required to provide the data and the number of models to try (max trials), and is returned a model which achieves the best performance (under the configured constraints) on that dataset. We can also specify the maximum model size, i.e the maximum number of scalars in the parameters of a model. Models larger than this are rejected during the search. This can be very useful for implementation on an edge-device.

We can either completely assign the task of model exploration to the AutoKeras framework, or we could atleast provide a rough-sketch or high-level architecture of the desired model. The earlier choice might return some arbitrary networks such as ResNet for image classification tasks, etc. Thus, it's beneficial to provide some clues to the framework regarding the model structure. In our Deep-Q network, we would ideally desire a simple architecture or, in other words, a Vanilla network. The input to our network is an image, and the output layer is a classification head (3 distinct actions). Such a model sketch can be provided to the AutoKeras framework using the following code snippet -

```
import autokeras as ak

input_node = ak.ImageInput()     # Image input

internal_node = ak.ImageBlock(
    block_type="vanilla",        # Vanilla network is desired
)(input_node)

output_node = ak.ClassificationHead()(internal_node)

# AutoKeras model is specified from input and output nodes
# Note the model constraints
model = ak.AutoModel(
  inputs=input_node, outputs=output_node, overwrite=True,
  max_trials=20, max_model_size=60000
)
```

Once we have specified the AutoKeras model along with various model constraints, the next step is to simply infer the network architecture search on the given dataset. This can be performed as follows -

```
# Train each model for 10 epochs during the search
model.fit(x_train, y_train, validation_split=0.15, epochs=10)
```

The search may take some time depending upon the number of trials and the epoch. Following image shows the console output during the training process -

```
Trial 15 Complete [00h 00m 11s]
val_loss: 0.13093478977680206

Best val_loss So Far: 0.10699634253978729
Total elapsed time: 00h 02m 48s
INFO:tensorflow:Oracle triggered exit
Epoch 1/10
75/75 [==============================] - 1s 8ms/step - loss: 0.2679 - accuracy: 0.9471
Epoch 2/10
75/75 [==============================] - 1s 8ms/step - loss: 0.2321 - accuracy: 0.9534
```

Once the training is completed, we can export the best model for further details as follows -

```
best_model = model.export_model()
best_model.summary()     # Shows the network architecture
```

7.8.3 AutoML in Reinforcement Learning

RL problems don't have a definite labelled dataset for training, rather they rely on their environment. There also exists a closed relationship between the model and the data. For example, the model is obviously influenced by the given input data at some timestamp. That's how the training is done. However, in the sequence alignment problem, the next pair of sub-sequences in a window will be decided by the current action predicted by the partially-trained model. Hence, the data is also influenced by the model. Thus, it becomes difficult to incorporate NAS in a situation where both data and model are dependent upon each other! In the previous training section [7.6], we have already trained the RL agent for various window sizes. Consider a window size of **50**, the best RL agent or deep Q-network is already available to us. Can we replicate this RL agent (window: 50) using AutoML with **lesser** parameters?

*We can start by collecting a sample of input states. This can be done by running the sequence alignment on any given pair of input sequences and saving the intermediate states (window of sub-sequences) separately. Further, we can use this best available network to generate a labelled dataset, which consists of predicted actions corresponding to each of the input states. This dataset might not be the ideal one, however it contains the predictions made by the current best model. For reducing the input dimension, we are representing the individual nucleotides using **2x2** pixel square. We have also removed the white empty space around the input image. Thus, we have modified the input states for dimensionality reduction.

Next, we can feed this generated dataset to the **AutoKeras** framework for finding an optimal model architecture with far lesser parameters. In theory, this optimal model should be able to replicate the current best model in terms of the output predictions. This process can be summarized using the following equations -

$$f(s,\theta) \longleftrightarrow AutoKeras \longleftrightarrow f'(s,\theta') \tag{7.8}$$

$$f(s,\theta) \approx f'(s,\theta') \;\; and \;\; |\theta'| << |\theta|$$

, where θ and θ' are the model parameters of the current best and optimal model, respectively. s represents the input states, f and f' represents the neural network functions for the current best and optimal model, respectively. Once we have obtained an optimal model (which is a reliable image of the current best model), we can further **fine-tune** it using our RL framework. This method can drastically reduce the model parameters, without much effecting the accuracy. The current best model (window: 50) had $\approx 13.6L$ parameters. This has been successfully reduced

to mere $\approx 98K$ parameters using AutoKeras. Regarding the accuracy, for one pair of sequences, the larger model gave 1311 matches and the smaller model gave 1290 matches. This smaller model can now be deployed on an edge-device. We also trained another AutoKeras model with window size of 70. The detailed network architecture, benchmarking results (Influenza dataset) and the inference timing (Jetson Nano) for the obtained optimal model is shown as follows -

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 100, 4, 32)        896

conv2d_1 (Conv2D)            (None, 100, 4, 32)        9248

max_pooling2d (MaxPooling2D  (None, 50, 2, 32)         0
)

conv2d_2 (Conv2D)            (None, 50, 2, 64)         18496

conv2d_3 (Conv2D)            (None, 50, 2, 32)         18464

max_pooling2d_1 (MaxPooling  (None, 25, 1, 32)         0
2D)

flatten (Flatten)            (None, 800)               0

dense (Dense)                (None, 64)                51264

dense_1 (Dense)              (None, 4)                 260

=================================================================
Total params: 98,628
Trainable params: 98,628
Non-trainable params: 0
```

Figure 7.19: Detailed architecture of the optimal model (window: 50). Note that it uses far lesser parameters to replicate the results of the current best model.

Window Size	**50**		**70**	
Model	**Previous**	**AutoKeras**	**Previous**	**AutoKeras**
% Mean Error	15.01	17.98	12.47	11.79
Inference Time (ms)	18.361	2.323	25.774	9.815
Model Parameters	1364739	98628	1758084	119108

Table 7.5: Benchmarking results on the Influenza dataset, Inference timing on the Jetson-Nano kit, and the Model parameters. Note that all the computations during the inference are performed in floating precision.

7.9 Comparison with existing implementations

We have also compared our work with some of the existing deep reinforcement learning-based implementations in the literature. Since none of them are deployed on the edge devices, a few comparison metrics such as the inference time (per action) is derived from their simulation results. The following table compares their performance with our EdgeAlign model for a window size of 50 -

	Type	Scoring Metric	Pre-processing	Model Size	Inf. time(ms)
Song et al. [18]	Pairwise	Exact matches	MUMmer	>13.8L	≈14
Jafari et al. [19]	Multiple	SP score	Not needed	>33K	NA
Ramchalam et al. [20]	Multiple	SP score	Not needed	>1.9L	NA
Joeres et al. [21]	Multiple	SP score	Not needed	NA	NA
Our Work	Pairwise	BLAST score	LCS	98628	2.323

Table 7.6: Comparison with existing implementations

, with the SP score defined as follows -

$$SP(A) = \sum_{i=1}^{N} \sum_{j=1}^{n-1} \sum_{k=i+1}^{n} sp\left(c_i^j, c_i^k\right) \quad \text{with} \quad sp\left(c_i^j, c_i^k\right) = \begin{cases} 0 & \text{if } c_i^j = c_i^k = - \\ 2 & \text{if } c_i^j = c_i^k \\ -1 & \text{if } c_i^j = - \text{ or } c_i^k = - \\ -2 & \text{if } c_i^j \neq c_i^k \end{cases}$$

Figure 7.20: Sum-of-pairs (SP) score

7.10 Deployment on Edge-devices

We have deployed the previously discussed AutoKeras model (window: 50) for the sequence alignment problem on 2 devices - STM32F746NG **Discovery** board, and the **Arty A7-100T** FPGA. First, we obtain the TFLite model file by following the steps mentioned in section 1.2.3. We need not apply any quantization as we would be using floating-precision model. Finally, we need to convert the TFLite model (.tflite) into a .h header file that can be loaded by TensorFlow Lite for Microcontrollers. The size of this header file is $\approx 389kB$. This is considerably larger as compared to the previous MNIST model, attributed to the fact that the model size is larger and it uses floating precision. Next, we will discuss the individual implementations.

7.10.1 Implementation on Disco board

We follow the steps mentioned in the chapter 2 regarding the generation of TFLite source files, configuring the application project and deployment. The clock speed is set at 100MHz. During the run-time, we first read both the sequences from the PC using an UART port. Next, we invoke the TFLite model and perform the sequence alignment. Finally, we send all the chosen actions back to the PC via the UART port. A Pyserial link has been setup on the PC interface to navigate this communication. An example application code is provided at main.cpp.

```
Length of sequence 1: 132
Length of sequence 2: 129

Reading sub-sequences...
Reading completed!
Starting sub-alignment...
Alignment completed!
Total actions: 138 | Duration(s): 214.850

Length of sequence 1: 1320
Length of sequence 2: 1323

Reading sub-sequences...
Reading completed!
Starting sub-alignment...
Alignment completed!
Total actions: 1352 | Duration(s): 2105.028

Total Matches : 1290
---------------------------------

Enter the FASTA file:
```

Figure 7.21: Deployment on Discovery board: Pyserial output during run-time.

Since the model file is large, we are storing it in the flash memory (constant array). This further increases the inference timing. Currently, the inference time for predicting a single action on the Disco board is **1.55 s** @100MHz. For some pair of sequence having length of 1500 each, the total alignment time is ≈ 38 min. This is considerably larger as compared to the Jetson inference timing (18.361 ms). The total size of the generated .elf file is ≈ **580 KB**.

text	data	bss	dec
453880	492	125248	579620

Table 7.7: Memory footprint of Sequence alignment application on Disco board

7.10.2 Implementation on Arty FPGA

We couldn't deploy our model on the CMOD A7-35T board since the model size is larger than the available BRAM (225KB). Hence, we deployed our RL model on a higher FPGA version of the same Xilinx family, Arty A7-100T board. This comes up with a 256MB DDR3 memory which is sufficient enough to hold any TinyML model. We didn't use BRAM to store the program data because we have reserved it's usage for the custom accelerator IPs discussed in the the next chapter.

LUT	Flip-Flop	Block RAM	DSP	DDR3
63400	126800	607.5 KB	240	256 MB

Table 7.8: Key specifications of Arty A7-100T board

DDR Configuration: All the program instructions and data are stored in the external DDR3 memory. The DDR3 is accessed via the Xilinx MIG (memory interface generator) IP. This facilitates in easily connecting the DDR to the microblaze core.

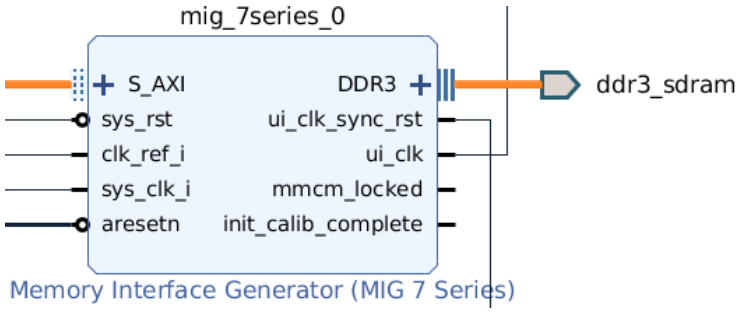


Figure 7.22: MIG IP connection with DDR

The MIG IP must be connected to the correct clock inputs in order to work properly. From the documentation, the ***clk_ref_i*** port must be connected to a **200 MHz** reference clock. The ***sys_clk_i*** should be connected to a **166.66 MHz** clock. ***sys_rst*** would be connected to the external system reset. Upon validating the design, the MIG IP automatically generates a reference clock of **83.33 MHz** at the ***ui_clk*** port. The rest of the microblaze design should ideally rely on this slower clock for it's functioning. This is yet another bottleneck while using the DDR memory, it slows down our design! Fortunately, we have bridges such as AXI Interconnect which can work with cross-domain clocks, i.e can handle IPs working with different clocks. Thus, the memory access can be performed using this slower clock and the rest of the design can operate at some higher clock speed (ex. 100MHz).

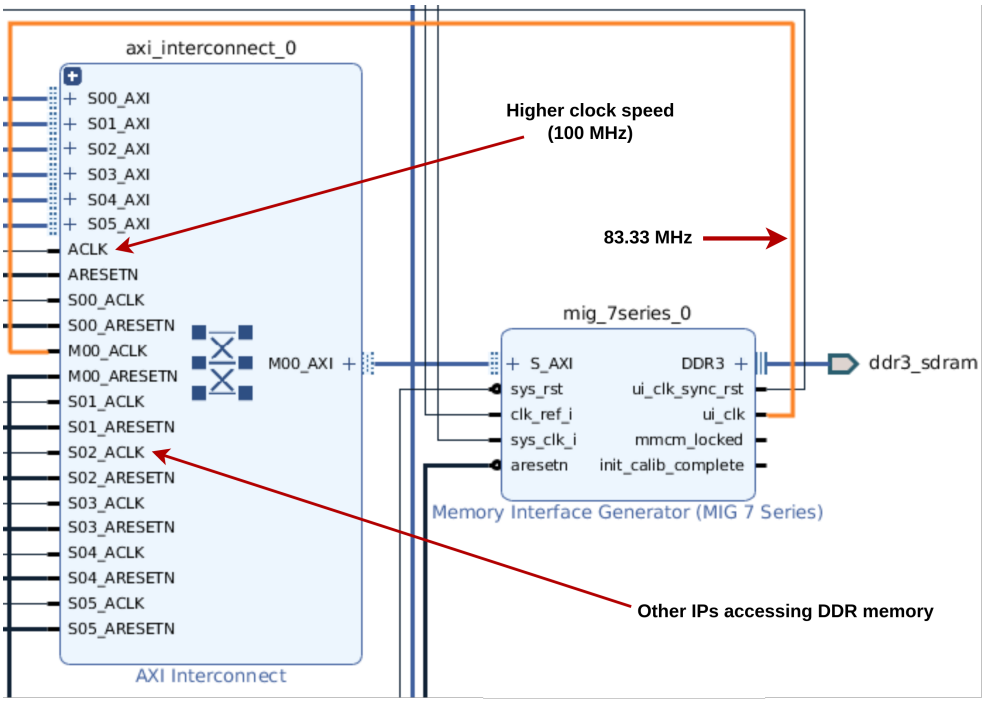


Figure 7.23: Higher clock configuration

The hardware configuration for the Arty board is a bit different as compared to what was discussed in chapter 4. We would be using the MicroBlaze soft IP as the CPU, however this time, we don't require to enable the AXI instruction interface. All the above clock inputs can be generated using the clocking wizard in vivado. Since the DDR memory access is slow, we would require instruction and data cache in microblaze for faster memory access.

Enable Instruction & Data Cache: We can enable the instruction and data cache in microblaze using its configuration window. We don't need to add a BRAM in the design as we did in the previous chapters. Instead, we can directly connect the *M_AXI_DC* (data cache) and *M_AXI_IC* (instruction cache) ports in microblaze to the MIG Interconnect. The base and high address in the data cache of microblaze must be carefully set to that of the DDR address.

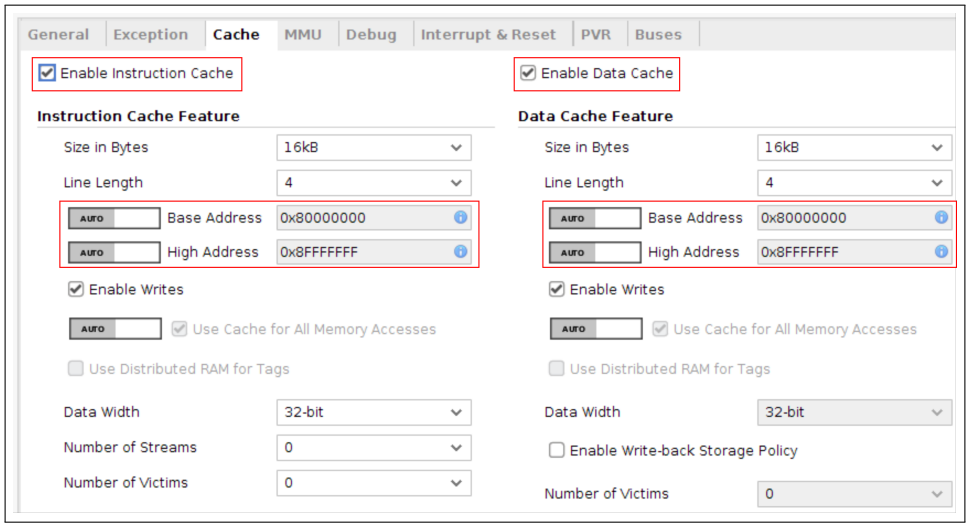


Figure 7.24: Cache configuration in MicroBlaze - Arty board

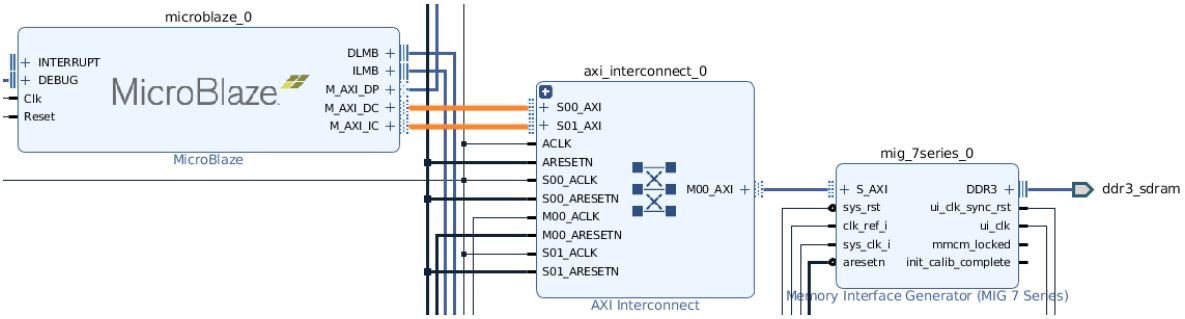


Figure 7.25: Instruction and Data cache connection in MicroBlaze

Lastly, we need to add few peripheral IPs in our design such as UART, Timer, etc. These peripherals would be operated using the *M_AXI_DP* port of microblaze via a separate AXI interconnect. Further, we can create the top-level HDL wrapper for our design and generate the bitstream. The final block diagram is shown as follows -

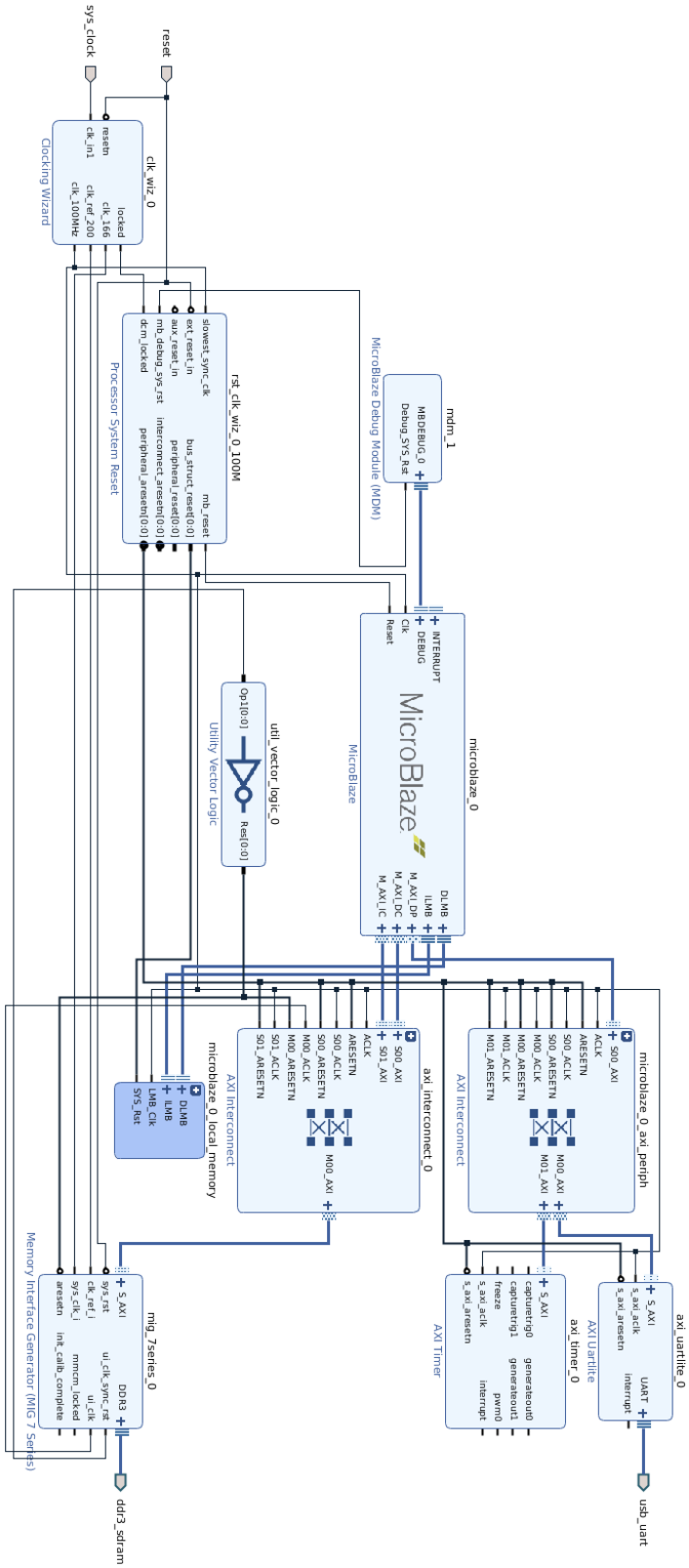


Figure 7.26: Block Design for MicroBlaze

Next, we can create an application project for our hardware platform. We follow the steps mentioned in the section 5.3 regarding the TFLite software development for FPGAs. The TFLite application code will be very similar to that of the ST board, with minor changes in the definition and initialization of the peripherals. An example application code is provided at main.cc.

Before we compile our application, we need to generate the Linker file for mapping different code sections at appropriate memory locations. Select the application project and go to Xilinx >Generate linker script. Specify the location of all Code, Data and Heap sections as DDR. Click Generate and compile the application to obtain the .elf file. To reduce the code memory, the Optimization setting in C++ build can be selected as Optimize for size (-Os). Next, we can proceed to run and benchmark our application.

The inference time for predicting a single action on the Arty FPGA board is **2.05 s** @100MHz. For some pair of sequence having length of 1500 each, the total alignment time is $\approx$ 52 min. This is considerably larger as compared to the Jetson kit as well as the ST board. This slower inference can be attributed to the usage of DDR and floating-precision computations. The total size of the generated .elf file is $\approx$ **750 KB**.

text	**data**	**bss**	**dec**
211556	399628	137960	749144

Table 7.9: Memory footprint of Sequence alignment application on Arty board

Since the current inference is extremely slow, we are required to design specialized IPs which will accelerate the computation of the deep neural network. Let's discuss the possible bottlenecks and design of DNN accelerators in the next chapter.

Chapter 8

CNN Hardware Accelerator

The previous chapter discussed about the pairwise sequence alignment application, its implementation and deployment on the edge-devices including FPGA. As evident from the inference timings, the run-time on the FPGA device is extremely slower as compared to its microcontroller counterpart (disco board) as well as the Jetson-nano kit. As we will further observe, CNN is at the core of our neural network computation. A majority of inference time is simply invested for computing the CNN output! Thus in this chapter, we will discuss about accelerating the CNN layer computation using specialized hardware blocks, or DNN accelerators. We will begin by introducing CNNs, why are they required to be accelerated, and our implementation of the accelerator IPs on FPGA.

8.1 Introduction

Convolutional Neural Networks (**CNN**) are often used for analysing patterns in the data, and is the most popular choice for visual applications. CNNs are regularized versions of multilayer perceptrons as they take advantage of the hierarchical pattern in data and assemble patterns of increasing complexity using smaller and simpler patterns embossed in their filters. Therefore, on a scale of connectivity and complexity, CNNs are on the lower extreme. They successfully capture the **Spatial** and **Temporal** dependencies in an image through the application of relevant filters.

CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns to optimize the filters (or kernels) through automated learning, whereas in traditional algorithms these filters are hand-engineered. This independence from prior knowledge and human intervention in feature extraction is a major advantage. But, how are CNNs computed?

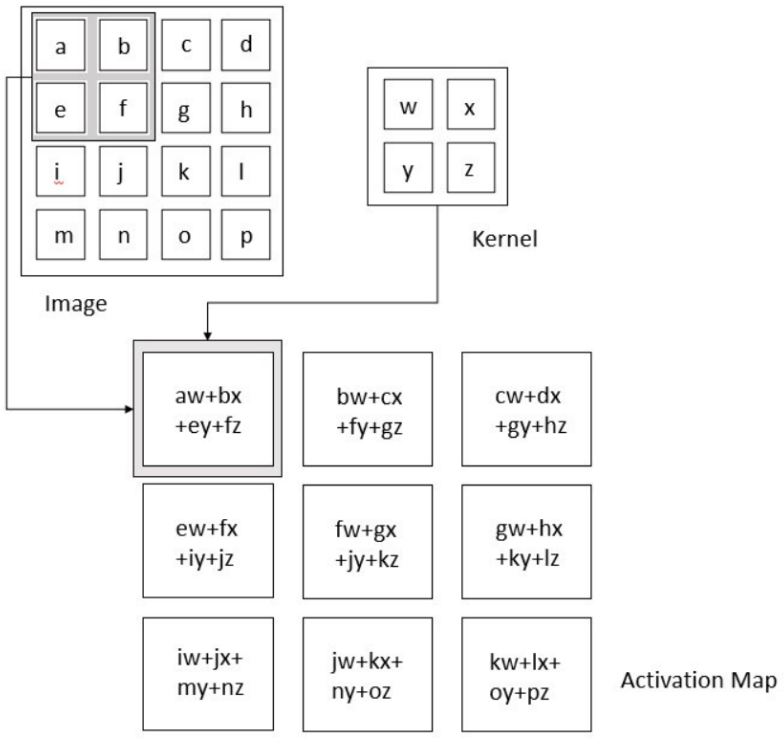
8.2 How are CNNs computed?

CNN layers are characterised by a mathematical operation known as **Convolution**. This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is a restricted portion of the receptive field. The kernel slides across the height and width of the image, thus producing an activation map which gives the response of the kernel at each spatial position of the image. The sliding size of the kernel is known as **stride**, i.e if the stride is 1, then we move the filters one pixel at a time.

The kernel is spatially smaller than an image but is more in-depth. This means that, if the image is composed of three (RGB) channels, the kernel height and width will be spatially smaller, but the depth extends up to all three channels. Mathematically, the 2D convolution operation can be expressed as follows -

$$G(x,y) = K * F = \sum_{i=-N}^{N} \sum_{j=-N}^{N} K(i,j)F(x+i,y+j) \tag{8.1}$$

, where K and F represents the kernel and input feature map, respectively. Thus, the convolution operation is very similar to a matrix-dot product, however, the data access pattern is non-uniform and often times, very complex (3D convolution)! Following figure shows an example convolution operation.



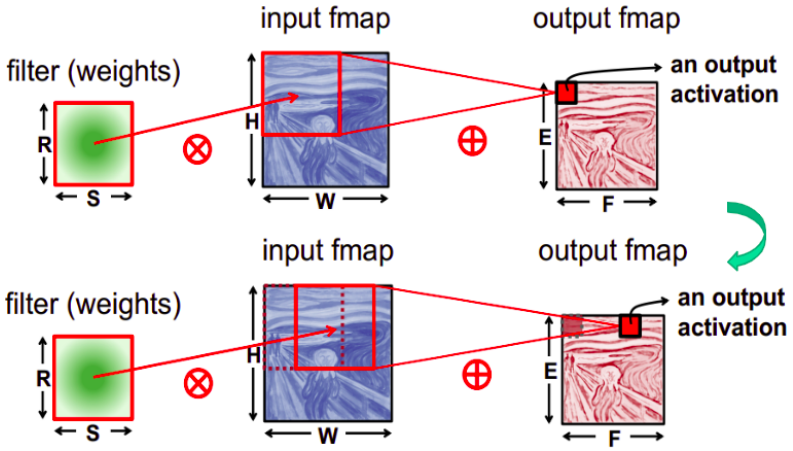


Figure 8.1: Convolution stride in action. Notice how the kernel moves.

Convolutional layers convolve the input and pass its result to the next layer. The kernel weights are learned during the training using backpropagation. These layers also help in dimensionality reduction. As evident from the above example, the dimension of the output activation map is smaller as compared to the input image. However, in few cases, we require the output dimension to be same as the input. In such scenarios, we can apply a valid **padding** to the input image before performing the convolution operation. On the other hand, 3D convolution are much more complex in terms of the data access pattern. We have several filters and each filter has some depth. The input image might also be available in several batches. The following figure highlights the 3D convolution operation -

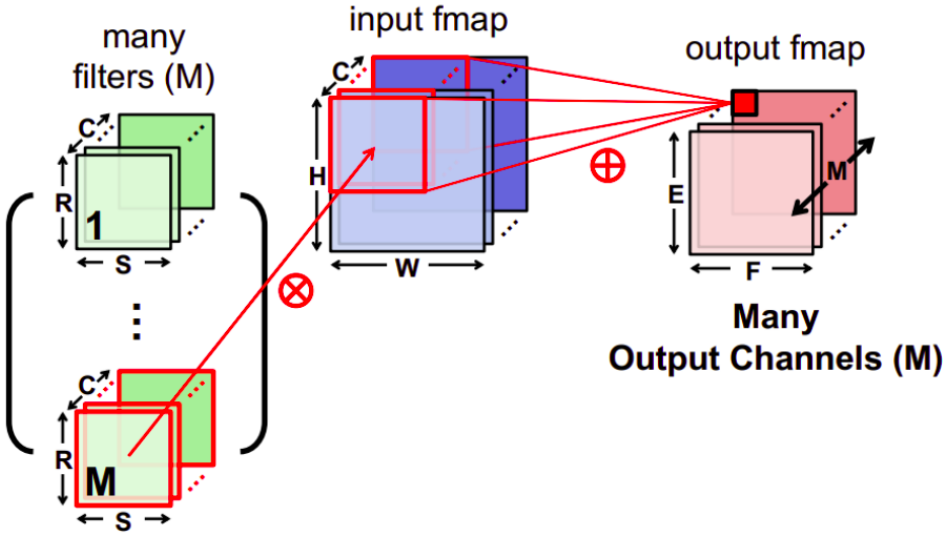


Figure 8.2: 3D convolution operation

*Another interesting approach for performing the CNN layer operations is via the matrix-vector multiplication. This is commonly used by the **GPUs** as they are specialized in such operations. We can re-order the input feature map to obtain a matrix **M** such that the convolution operation can be converted to a matrix-vector multiplication operation as follows -

$$F^{N\times N} * K^{M\times M} = M^{O^2\times M^2} \times K^{M^2 x1} \quad (where\ O = N - M + 1\ ,\ if\ stride = 1)$$

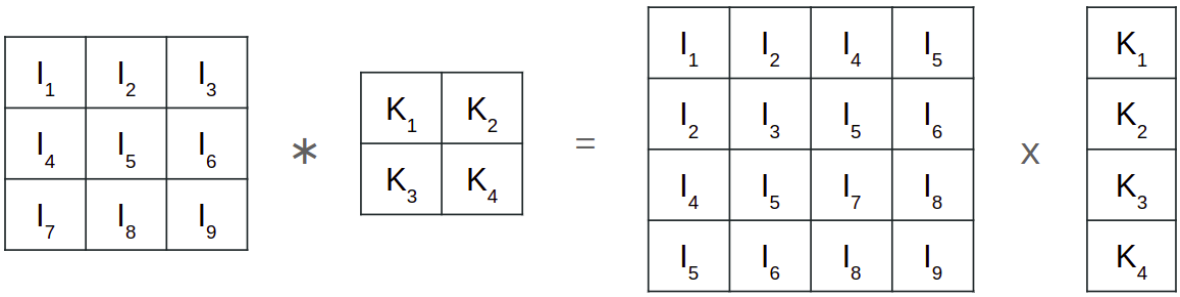


Figure 8.3: Convolution operation using Matrix-vector multiplication

The size of this matrix M can be very large and it's not scalable to be implemented on an edge-device. However, this might re-use any specialized hardware block which performs the matrix-vector multiplication.

These convolution layers can be stacked up together along with few fully connected layers at the end to form a complete network. The following figure shows an example network for classifying handwritten digits -

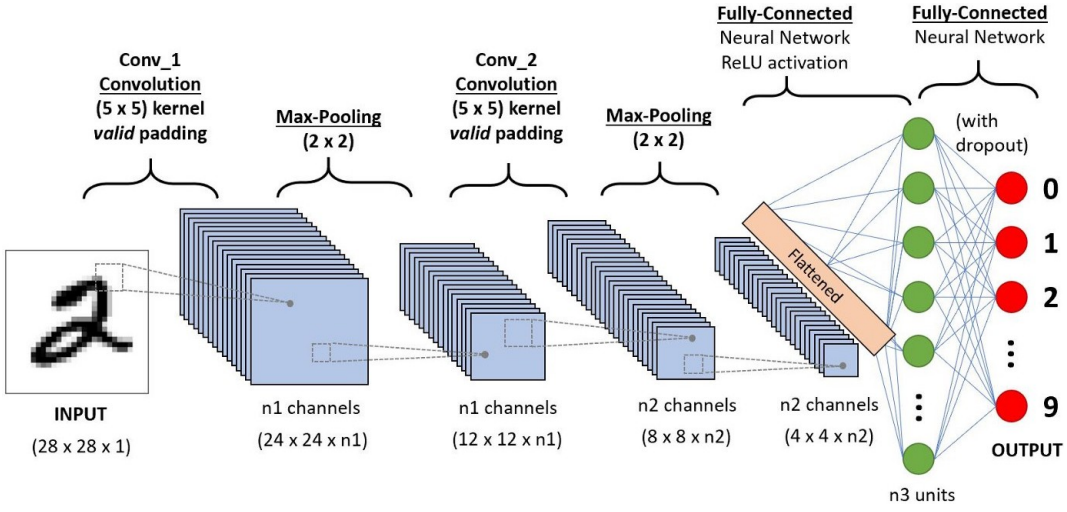


Figure 8.4: A CNN sequence to classify handwritten digits

8.3 Why accelerate CNNs?

CNNs might appear to be very simple networks with fewer parameters. However, there are a few bottlenecks when it comes to their deployment on the hardware devices. These often translate to an increased inference timing and energy consumption. The purpose of a CNN hardware accelerator is to tackle these issues and improve performance. Let's discuss them in detail.

8.3.1 Number of Operations

Let's compare the total number of MAC operations required for computing the first FC layer, and the second CNN layer output in our AutoKeras model.

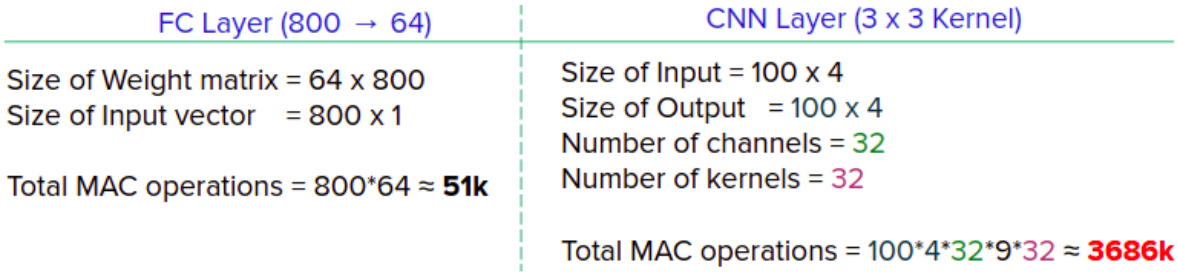


Figure 8.5: Comparison between number of required MAC operations

Layer	# Parameters	# MAC operations
FC	51264	51200
CNN	9248	3686400

The CNN layer is around ≈ 70 times expensive as compared to the FC layer in terms of the computation! However, the number of kernel parameters is much lesser in CNNs. Thus, the CNN layers are spatially efficient and computationally inefficient. Even a very small kernel could result in large number of MAC operations.

A sequential hardware would be very expensive in such a scenario and would result in long inference timings. Thus, the CNN accelerator must incur parallelism in the hardware design. As we discussed in the previous chapters regarding the design of custom IPs, several MAC operations can be performed in parallel using an appropriate hardware topology. However, the complex and non-uniform data access pattern in CNN often poses a challenge for such designs.

8.3.2 Memory Access

All the model weights and biases are usually stored in some external memory such as DRAM, etc. During the computation, the processing hardware block simply fetches the required operands from the memory, performs the required computation and stores the result back into the memory. Now, let's have a look at the energy cost data acquired from a commercial 65nm process.

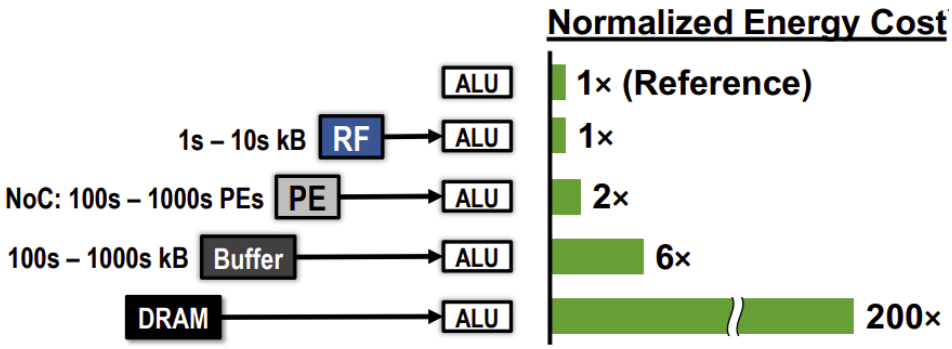


Figure 8.6: Normalized energy cost measured from a commercial 65nm process

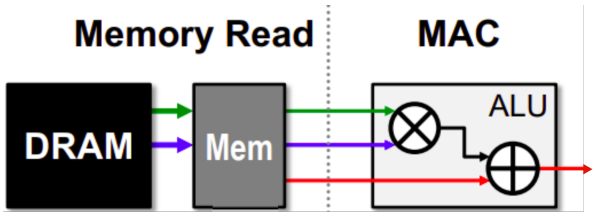
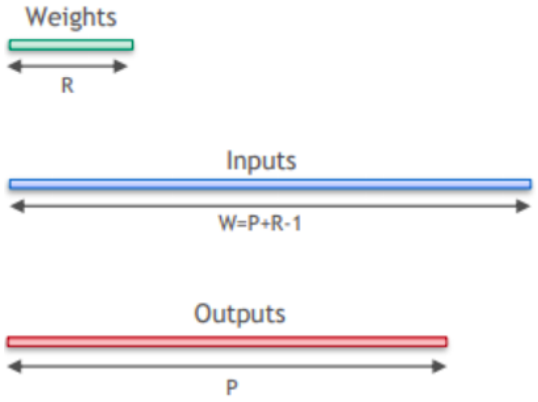


Figure 8.7: Local memory buffer for data reuse

Fetching data from a larger memory is **expensive** both in terms of energy as well as the access time. Usually, designers also use few additional local memory blocks in the design in-order to prevent repetitive direct access. These local memory are smaller, but faster and more energy-efficient. However, what if in the worst case, all memory R/W are DRAM accesses. Thus, memory access is the bottleneck.

This problem is further amplified while using the convolution operation where we are required to read a single operand multiple number of times. In such scenarios, data reuse can significantly improve the performance of the system. We can use few highly energy-efficient and faster, yet smaller memory blocks such as register files for storing and processing the intermediate results. Let's have a look at few dataflow techniques which maximises the data reuse with the low-cost memory hierarchy and parallelism.

Let's have a look at the following 1D convolution example. The size of input, weight and output vector are W, R and P, respectively.




```
for r = [0:R)                                  # Iterate through all weights
    for p = [0:P)                              # Iterate through all outputs
        Output[p] += Weight[r] * Input[p+r]    # MAC operation
```

As a software developer, we might simply write the above code which performs the required 1D convolution. However, observe that the weight operand ($weight[r]$) remains **constant** for all the iterations inside the inner loop. Thus, instead of repetitively fetching the same weight from the memory, we can store it in some local variable (register file), and reuse it for further computations as illustrated in the following code. This not only improves the energy-efficiency but also reduces the computation time. This method is known as **weight-stationary** approach.

```
for r = [0:R)                                  # Iterate through all weights
    w_local = Weight[r]                        # Local variable
    for p = [0:P)                              # Iterate through all outputs
        Output[p] += w_local * Input[p+r]      # MAC operation
```

Similarly, we can also have Input and Output stationary implementations which minimizes the data movement of input and output operands, respectively. Following code illustrates the output-stationary implementation.

```
# Output-stationary implementation
for p = [0:P)
    out_local = Output[p]                      # Local variable
    for r = [0:R)
        out_local += Weight[r] * Input[p+r]    # MAC operation
    Output[p] = out_local                      # Write-back
```

8.4 Row-Stationary Convolution

The previous section highlighted the significance of parallelism and data reuse in the convolution operation. We discussed about different approaches for maximizing the data reuse. Another such brilliant idea is the Row-stationary implementation which maximizes the row convolutional reuse at the register file level, i.e keeping a row of the filter and feature map constant. Let's take an example of the 1D convolution.

We can synthesize a specialized hardware block known as Processing Engine or **PE**, which computes the convolution of 1-feature row and 1-kernel row. All the input and kernel elements (entire row) are stored in a local buffer associated with the PE. This PE also consists of shift registers for storing and processing the feature and kernel elements. During the computation of the an output element, all the relevant input and kernel elements are fetched and stored inside these shift registers. Further, we have a multiply-and-accumulate block which performs the MAC operation and stores the result in another register. Once the computation for the current output element gets completed, the input register file is simply shifted to expel the unnecessary elements and fetch the new ones. This process is repeated until all the output elements are computed. Thus, for calculating a single row of the output, the entire row of the kernel and input feature map is held constant and only fetched once from the external memory. Hence, this method is known as row-stationary.

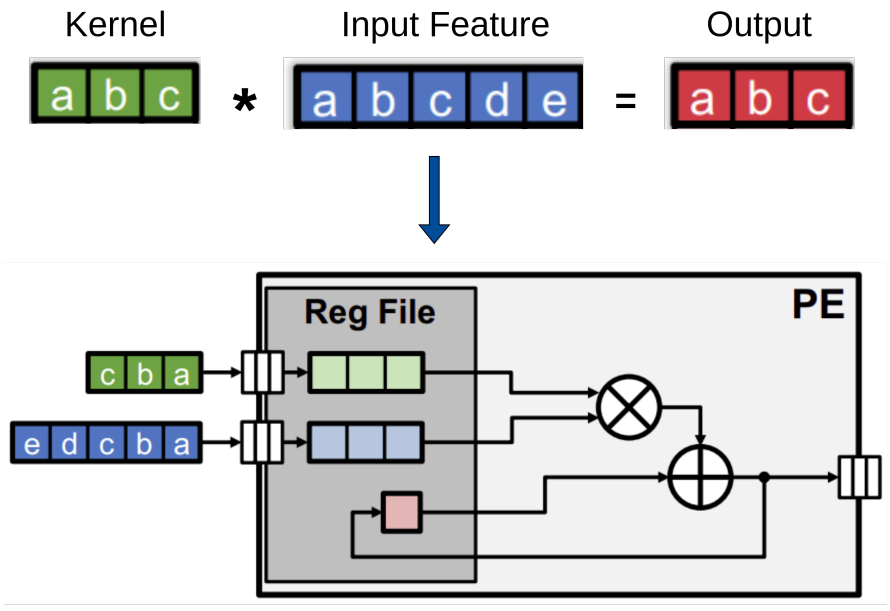


Figure 8.8: Hardware realization of Processing Engine

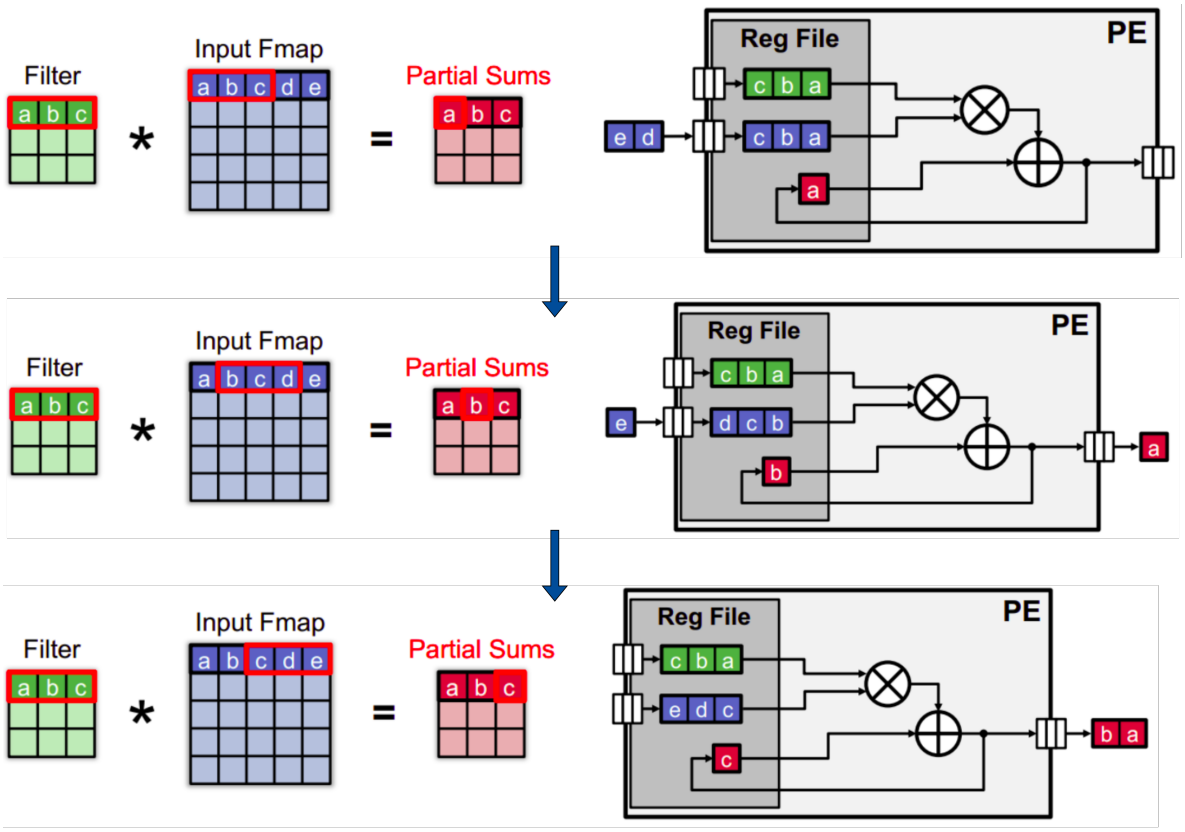


Figure 8.9: Working of the Processing Engine

The above figure shows the complete working of the PE during the 1D convolution of input and kernel row. How do we scale this design for 2D convolutions? Observe that a particular output row in the 2D convolution is composed of various partial convolutions, each of which can be computed using the above PE. For example, if the stride is 1 and there isn't any padding, the i^{th} output row (O_i) can be calculated as follows -

$$O_i = \sum_{j=1}^{N} PE_{j,\, i+j-1}$$

, where $PE_{i,j}$ represents the convolution output for the i^{th} kernel row and j^{th} input row calculated using the above PE. N represents the number of kernel rows. Thus, we can use several PEs in our design, each of which can operate in **parallel**. For computing a single output row, we would require N number of PEs. These PEs must be connected with each other using a data bus, so that they can pass-on their relevant data such as partial convolution products to the neighbouring PEs for accumulation.

Such an architecture is illustrated by the following figure for a kernel size of 3x3 and input feature size of 5x5.

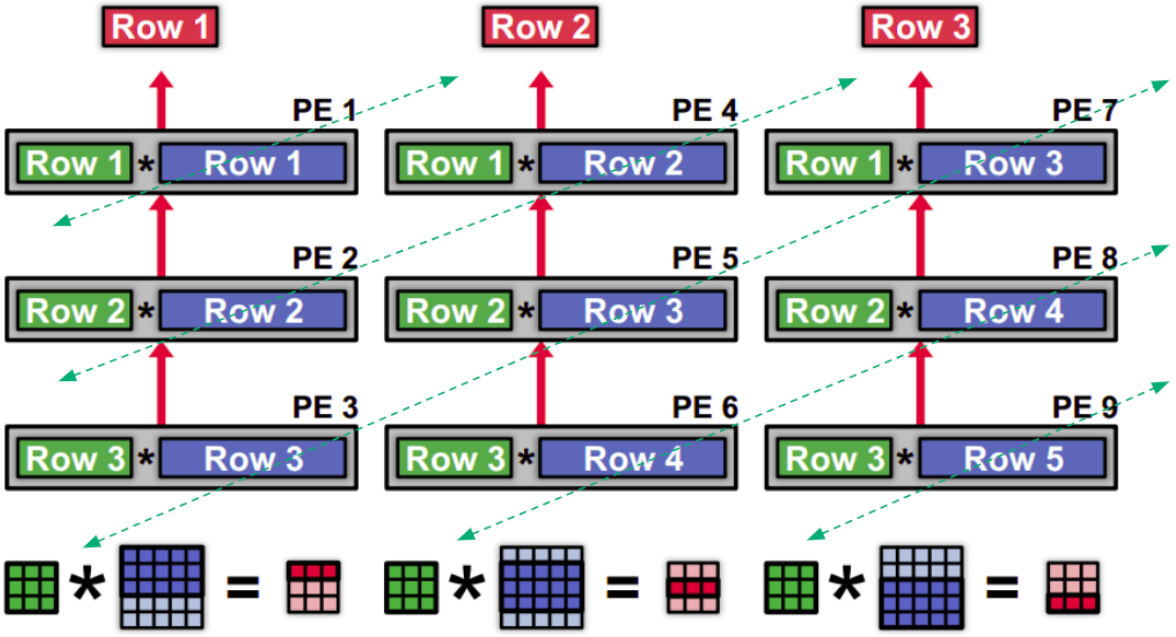


Figure 8.10: 2D Convolution using PE Array

Notice that the filter rows are reused across PEs horizontally, the input feature map rows are reused across PEs diagonally, and the partial sums are accumulated across PEs vertically. This significantly improves the parallelism and data reuse in our design. In a 3D convolution, we can simply store the row values in an interleaved fashion. We will discuss this more in detail later.

This architecture is also used by the Eyeriss hardware [41], which is an energy-efficient reconfigurable accelerator for deep convolutional neural networks. They even have their own optimized compiler for mapping the neural netowrk architecture over the Eyeriss chip. More details regarding the same can be acquired from the paper and their website https://eyeriss.mit.edu/.

A more challenging task is to implement and benchmark such an architecture using the available design tools. For our application, we have used **Vivado HLS** for synthesizing the above design. Further, we create a convolution IP out of it and integrate it with our existing FPGA hardware. The design procedure would be similar to section 5.4, however in this case, the HLS part is a bit more challenging. In the next section, we will discuss about various designs that we tried, several optimization pragmas and design techniques.

8.5 Designing the CNN Accelerator

We have tried out various hardware designs before arriving at the best one. All the designs are based on the PE array architecture as discussed in the previous section. However, these can be differentiated based on their data delivery system. We have used **Vivado HLS 2021.2** for synthesizing our designs. In this section, we will discuss each of them individually, their performance and drawbacks. Let's first begin with a basic design which will act as a reference for estimating performance gains.

8.5.1 Design-I : Sequential

This is the most trivial hardware design for performing the convolution operation. All the kernel and input feature elements are read from the external memory, and stored in a local buffer. Further, the MAC operations are performed sequentially and the results are stored back into the memory. The design architecture and corresponding HLS code can be summarized as follows -

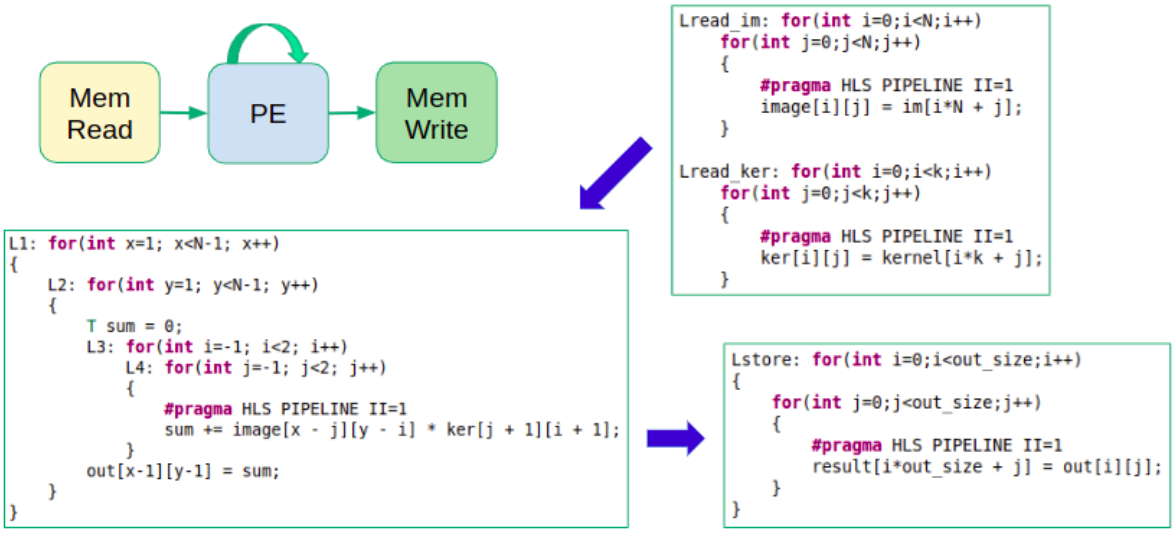


Figure 8.11: Design-I: Sequential operation

The following figure shows the resource utilization and co-simulation latency values for (N = 5, K = 3) & (N = 15, K = 3) configurations. N represents the dimension of the input feature map (NxN), and K represents the kernel dimension (KxK). This design is much slower considering the average time required to perform a single MAC operation. Next, we will discuss about a PE array design which uses global buffer as a data delivery system.

N = 5, K = 3

BRAM_18K	DSP	FF	LUT	Co-sim Latency
2	3	3048	2974	313

N = 15, K = 3

BRAM_18K	DSP	FF	LUT	Co-sim Latency
4	2	2494	2703	3130

Figure 8.12: Design-I: Resource Utilization and Latency

8.5.2 Design-II : Global Buffers

This design consists of a 2D array of PEs, each of which is working in parallel to compute the partial convolution products. In the vertical direction, each of the PEs in a column are connected using an unidirectional streaming interface. This stream is used to communicate the partial convolution output to the next PE.

On the other hand, this design uses various common global buffers for delivering the kernel and input elements to different PEs. For instance, all the PEs in a row use a common array buffer for accessing the corresponding kernel row elements. Similarly, all the PEs in a diagonal use a common buffer for accessing the corresponding input row elements. This design incurs task-level parallelism by using the *Dataflow* HLS prgama. Since we are using arrays in the design, it is easier to implement and debug. Figure 8.13 summarizes the design architecture for (N=5, K=3).

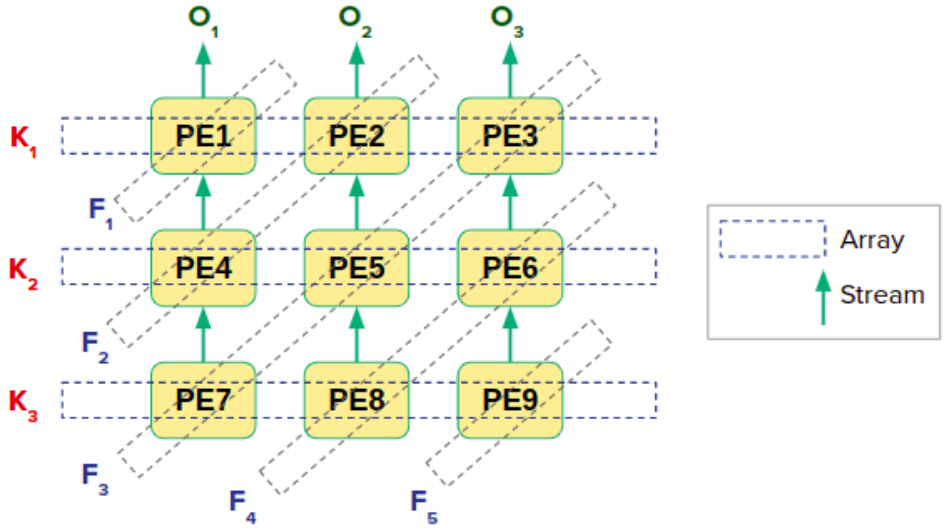


Figure 8.13: Design-II: Global Array + Stream Interface

Notice the resource utilization and co-simulation latency for this design. As evident, this design is much slower as compared to the reference design-I. This can be attributed to the fact that it uses array interfaces and hence, multiple read/writes are not allowed at the same instance. Although the PEs are working in parallel, the intermediate memory access is almost sequential. Thus, this design offers low-bandwidth memory access and deteriorates with scale.

BRAM_18K	DSP	FF	LUT	Co-sim Latency
6	3	4656	5712	538

N = 5

BRAM_18K	DSP	FF	LUT	Co-sim Latency
7	3	4297	8775	4228

N = 15

Figure 8.14: Design-II: Resource Utilization and Latency

We have used various pragmas in our HLS code, including *Pipeline*, *Unroll*, *Dataflow*, etc. Out of these, the *Dataflow* pragma is the most important as it enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation, and increasing the overall throughput of the design. More details can be gathered from the official Xilinx documentation. Next, we will discuss about another PE array design which uses only streaming interfaces for data delivery.

8.5.3 Design-III : Streaming Interfaces

This design is similar to the previous one, except that it only uses streaming interfaces for data delivery. Each PE sends the required operands to the neighbouring PE (both diagonally and horizontally) using streams. The kernel elements are streamed in the horizontal direction, whereas the input feature elements are streamed in the diagonal direction. For the entry-level memory access, the input and kernel elements are only read at the boundaries of the PE array. In the vertical direction, each of the PEs in a column are again connected using a streaming interface which is used to communicate the partial convolution output to the next PE. Figure 8.15 summarizes the design architecture for (N=5, K=3).

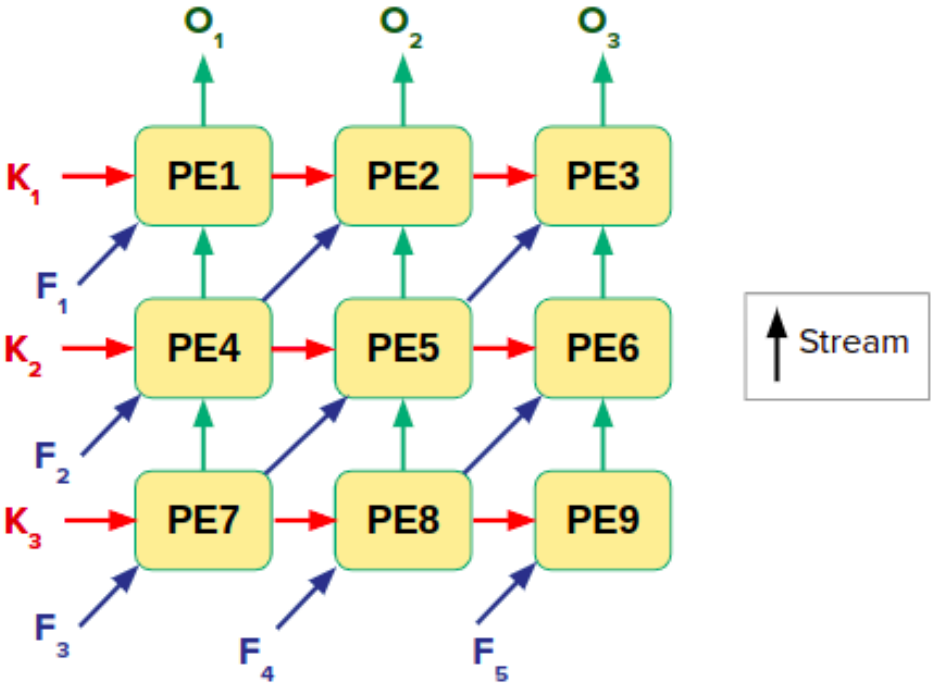


Figure 8.15: Design-III: Streaming Interfaces

O_i, K_i and F_i represents the i^{th} output, kernel and input feature row, respectively. Thus, each PE is responsible for **forwarding** the required operands to the neighbouring PE for further computation. These streaming interfaces are implemented as FIFOs in the hardware and they are extremely fast.

*Since all the stream connections are hardwired, this design is not flexible! What if the stride changes? In such scenarios, we need to modify the interconnections and re-synthesize the hardware. The following figure shows the resource utilization and co-simulation latency. This design is much faster as compared to our reference design-I, along with much higher resource utilization. Refer the HLS code for details.

BRAM_18K	DSP	FF	LUT	Co-sim Latency	
24	9	5563	7576	376	N = 5

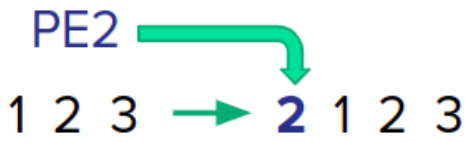
BRAM_18K	DSP	FF	LUT	Co-sim Latency	
84	12	7533	15977	1987	N = 15

Figure 8.16: Design-III: Resource Utilization and Latency

8.5.4 Design-IV : Routing Interfaces-I

The previous design wasn't flexible because the interconnections between the individual PEs were hardwired. In this design, we assign the responsibility of delivering the data to specialized hardware blocks called **Routers**. These routers can identify the suitable PEs by accounting the adjustable model parameters such as stride, etc. and can further deliver the required data. This is very similar to how internet works. Each PE has an address which is very similar to an IP address in internet. Each data packet would be appended with a header value, which is required by the routers for routing the data packet to the correct address.

Let's take an example. We wish to deliver a vector with data elements (1,2,3) to **PE2**. We can append a value of **2** at the start of this data packet as shown by the following figure. Here, 2 signifies the address of the corresponding PE. This header value would be further used by different routers for routing this data packet. Figure 8.17 summarizes the design architecture for (N=5, K=3). Thus, the path taken by the above data packet would be: **R0 ->R1 ->PE2**. Such a netowrk resembles a Network-on-chip (**NoC**).



Regarding the entry level memory access, all the inputs (kernel and input feature elements) would be streamed into the PE array using a common streaming interface (*Stream_in*). The headers would be inserted at the entry-level stage. In the vertical direction, each of the PEs in a column are again connected using a streaming interface which is used to communicate the partial convolution output to the next PE.

However, due to complex routing logic, we were not able to synthesize and benchmark the design. Still, this design provides a good head start towards the development of flexible CNN accelerator designs. The similar intuition would be carried forward in the future designs. Next, we will discuss about another design which involves routing interfaces.

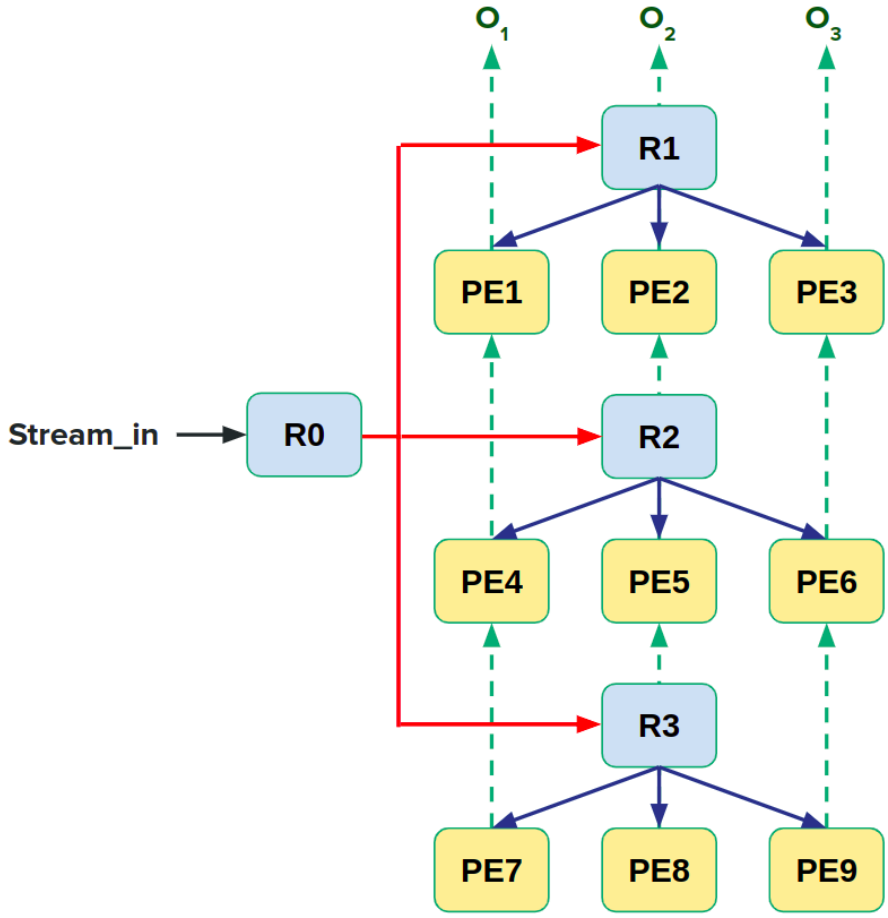


Figure 8.17: Design-IV: Routing Interfaces-I

8.5.5 Design-V : Routing Interfaces-II

In this design, each PE has its own dedicated router. All the input operands (kernel and feature elements) are streamed to respective rows of PEs. Based on the header value, the individual Routers R(i) can either **forward** the received data to the next router, or **pass** them to the corresponding PE. Thus, the visibility of the PEs towards the incoming data is controlled by their routers.

For the last PE in a row to receive the data operands, it needs to wait for all the previous routers in that row to forward it until it reaches its own router. Such a data delivery system would be extremely slower and unscalable. We were able to synthesize this design, however, the co-simulation failed for some reasons (complexity, etc.). Yet, we were able to obtain the resource utilization and process timings as shown in figure 8.18. For example, the routing process itself takes around **4.5K** cy-

cles for N=15 configuration! Thus, this design is much slower as compared to our reference design-I. Figure 8.19 summarizes the design architecture for (N=5, K=3).

		Latency		Interval	
Instance	Module	min	max	min	max
Loop_Lpe1_proc19_U0	Loop_Lpe1_proc19	268	268	268	268
Loop_Lrout1_proc18_U0	Loop_Lrout1_proc18	4486	11038	4486	11038
read_memory_U0	read_memory	?	?	?	?
Loop_3_proc_U0	Loop_3_proc	176	176	176	176
conv_pe_entry9_U0	conv_pe_entry9	0	0	0	0

Figure 8.18: Design-V: Process Timings (N=15)

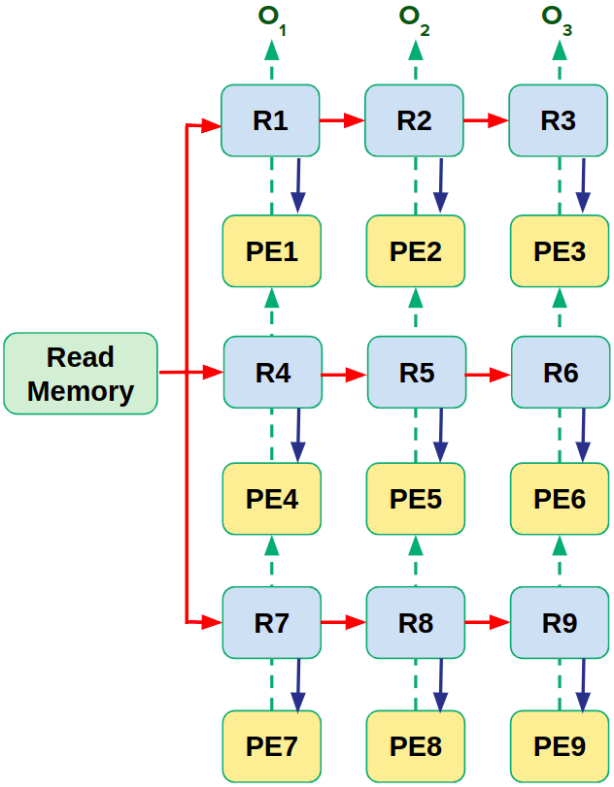


Figure 8.19: Design-V: Routing Interfaces-II

	BRAM_18K	DSP	FF	LUT
N = 15	6	39	11070	22455

Figure 8.20: Design-V: Resource Utilization

8.5.6 Design-VI : Selective Routing

In this design, each router is assigned to route only a single data row to the suitable PEs. We have Kernel routers for routing the individual kernel rows to the required PEs. Similarly, we have Feature routers for routing the individual feature map rows to the suitable PEs. The individual PEs are only connected in the vertical direction using the streaming interface. Since, each router works with a single data row only, hence the name is Selective routing.

This design offers the best of both the worlds, i.e Flexibility and Direct data delivery. The required PEs can be identified by accounting the adjustable model parameters such as stride, etc. The total number of routers required in this design would be **N + K**, where N and K represents the number of feature and kernel rows, respectively. Figure 8.21 shows the design architecture for feature routers. Once the individual feature rows are read from the memory, they are streamed to appropriate feature routers for further streaming. These routers then forward their individual rows to the suitable PEs. Similarly, figure 8.22 shows the design architecture for kernel routers.

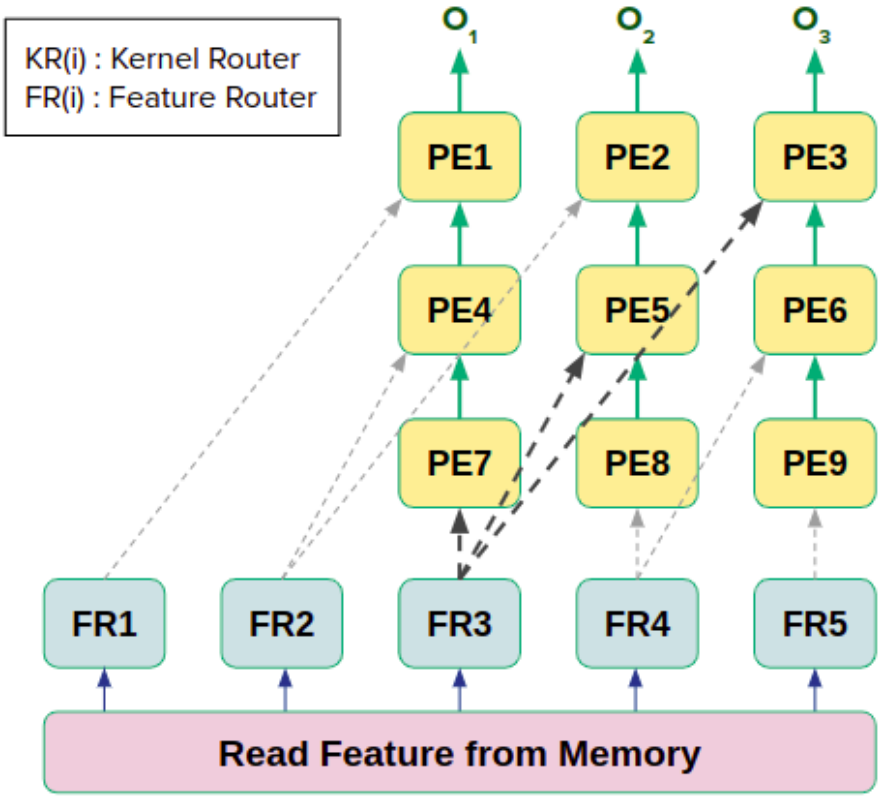


Figure 8.21: Design-VI: Feature routers for routing feature rows

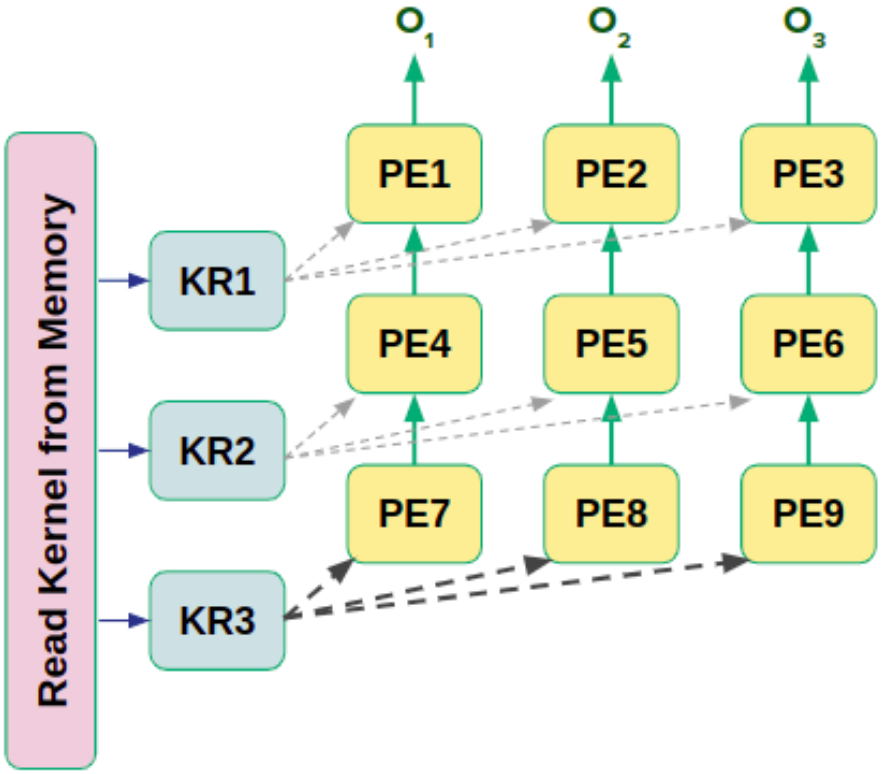


Figure 8.22: Design-VI: Kernel routers for routing kernel rows

Figure 8.23 shows the resource utilization and co-simulation latency for this design. It is much faster as compared to our reference design-I, however, a bit slower as compared to the design-III. A slight increase in the latency is observed at the expense of higher flexibility. Yet, this is the best design so far in terms of the routing efficiency. Refer the HLS code for further implementation details. Next, we will discuss about a design which uses matrix-multiplication for performing convolution.

BRAM_18K	DSP	FF	LUT	Co-sim Latency
6	9	5483	8484	430

N = 5

BRAM_18K	DSP	FF	LUT	Co-sim Latency
6	40	10598	22331	2167

N = 15

Figure 8.23: Design-VI: Resource Utilization and Latency

8.5.7 Design-VII : Convolution using Matrix-Multiplication

This design uses the matrix-multiplication operation for performing convolution. As discussed earlier, the input feature map is re-ordered to obtain a matrix which when multiplied with the kernel vector produces the convolution output. All the kernel and input feature elements are read from the external memory, and stored in a local buffer. Further, a matrix-multiplication is performed as the final step. The corresponding HLS code, resource utilization and latency is as follows -

```
Lread_im: for(int x=0;x<N;x++)
    for(int y=0;y<N;y++)
    {
        #pragma HLS PIPELINE II=1
        T v = im[x*N + y];
        for(int i=-1; i<2; i++)
            for(int j=-1; j<2; j++)
            {
                #pragma HLS UNROLL
                x_t = x + j -1;
                y_t = y + i -1;
                if(x_t <out_size && x_t>=0 && y_t<out_size && y_t>=0)
                    fea[x_t*out_size + y_t][(j+1)*k +i+1] = v;
            }
    }
```

```
Lread_ker: for(int i=0;i<k;i++)
    for(int j=0;j<k;j++)
    {
        #pragma HLS PIPELINE II=1
        #pragma HLS LOOP_FLATTEN
        ker[i*k + j] = kernel[i*k + j];
    }
```

```
Lmult1: for(int j=0;j<(k*k);j++)
    {
        T v = ker[j];
        Lmult2: for(int i=0;i<(out_size*out_size);i++)
        {
            #pragma HLS PIPELINE II=1
            T prev = (j==0)? static_cast<T>(0): out[i];
            out[i] = prev + fea[i][j] * v;
            #pragma HLS DEPENDENCE false variable=out
        }
    }
```

Figure 8.24: Design-VII: Convolution using Matrix-Multiplication (HLS)

BRAM_18K	DSP	FF	LUT	Co-sim Latency
8	3	4513	4741	376

N = 5

BRAM_18K	DSP	FF	LUT	Co-sim Latency
11	4	3956	4515	3022

N = 15

Figure 8.25: Design-VII: Resource Utilization and Latency

8.5.8 Conclusion

We have discussed various CNN accelerator designs in this section, along with their performance and resource-utilization. However, since our application of pairwise sequence alignment is very targeted, we will proceed with the **Design-III** which involves direct-communication among PEs. It performs well both in terms of resource-utilization and latency. For future applications, we might go forward with the Design-VI as it is more flexible.

We also modified the design III for performing 3D convolution with variable system parameters such as image dimension, etc. Even the main loop in the dataflow was pipelined to maximize the task-level parallelism! However, it resulted in a very high resource-utilization (ex, LUT ≈1L) and thus, it wasn't possible to implement it on edge-devices such as the Arty board.

In order to perform the 3D convolution, we took help of the MicroBlaze. A 3D convolution is essentially composed of various 2D convolution results, each of which is accumulated together to obtain the final result. Thus, we made a custom IP based on the design III which performs the 2D convolution. On the MicroBlaze side, this IP is invoked iteratively and all the results are added together to obtain the final convolution products. Such an approach uses little resources and can be implemented on edge-devices. The following figure shows a dummy code for the same. Next, we can proceed to utilize these IPs in our application.

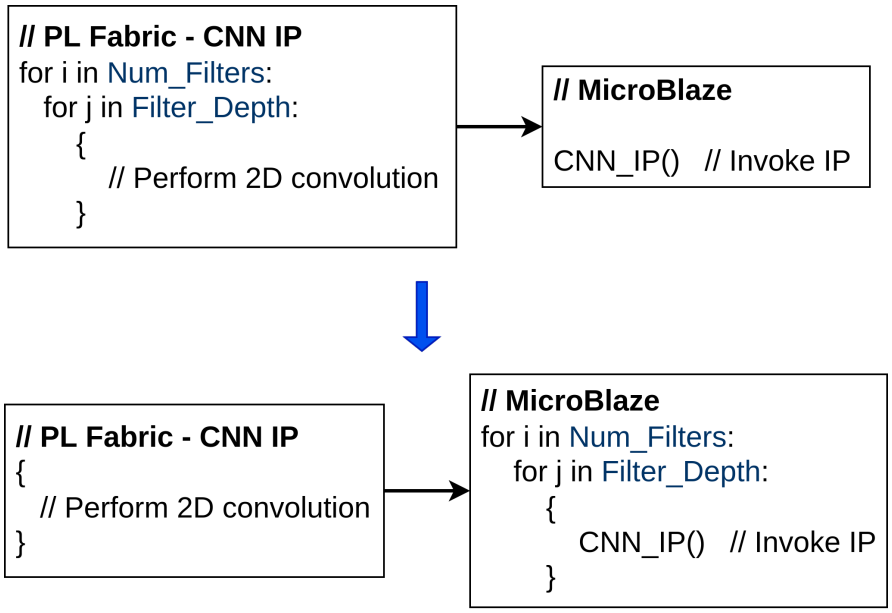


Figure 8.26: 3D convolution using MicroBlaze and CNN IP

8.6 Integration with TFLite Application

In this section, we will discuss about how to integrate the CNN accelerator IP with our Vivado hardware design and TFLite application. This process would be similar to section 5.4. We will begin by creating and exporting the required custom IPs, followed by modifying our vivado design and TFLite kernel. The Vivado version used is 2021.2.

8.6.1 Creation of Custom IPs using Vivado HLS

Our AutoKeras model (window: 50) uses 2 different types of CNN layers. These layers use 2D input dimensions of **100x4** and **50x2**, respectively. Since, we are using a hardwired architecture, we need 2 different IPs corresponding to these different layers. We have also accounted the padding operation and bias in convolution layers during the creation of these IPs.

The kernel, image and bias elements are read from a memory-mapped AXI interface (***m_axi***). The output data is also written on a m_axi interface. Thus, we can treat these inputs as array pointers. Rest of the scalar inputs are declared as AXI4-Lite interface (***s_ axilite***). Once the design is finalized, we can synthesize and implement our Custom IPs. Timing violations must be checked, if any. Table 8.1 shows the resource-utilization and HLS code for these different IPs. Finally, we can export the RTL to integrate it in our Vivado project.

Layer Input	BRAM_18K	DSP	FF	LUT	HLS Code
100 x 4	74	57	11928	14710	CNN_100x4.cpp
50 x 2	32	27	6433	8403	CNN_50x2.cpp

Table 8.1: Resource utilization and HLS code for the CNN accelerator IPs

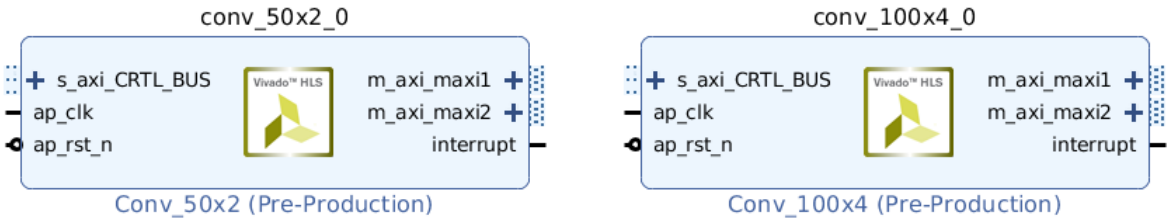


Figure 8.27: CNN Accelerator IPs

8.6.2 Integration in Vivado

The integration of these IPs in our Vivado block design is similar to the earlier sections. However, in the case of Arty board, we can directly connect the *m_axi* ports with the MIG AXI Interconnect to access the data memory of microblaze. These IPs are driven by the same clock as the MicroBlaze. The final connection should appear as follows -

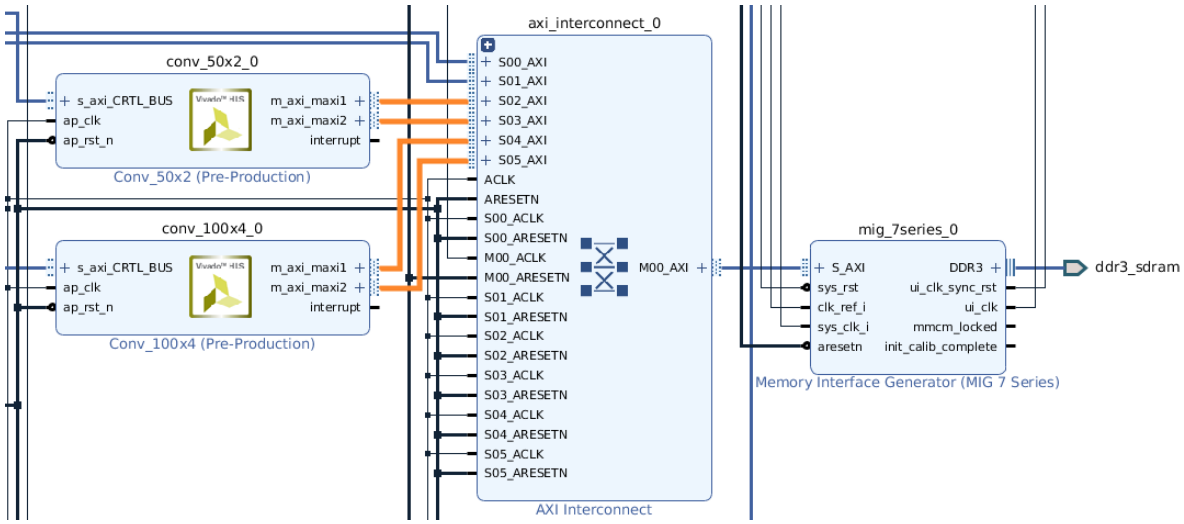


Figure 8.28: Integration with hardware design

If the design consists of more than one Custom IP, then we can simply increase the number of slave ports of the MIG AXI InterConnect, and connect them with the individual custom IPs. More number of custom IPs would reduce the available data bandwidth and might slow down the data access. Once the design is validated, we can generate the bitstream and export the hardware. The following table shows the resource utilization of the final design involving the custom accelerator IPs -

BRAM_36K	DSP	Flip-Flop	LUT
51	89	29012	25513

8.6.3 Modifying the TFLite Kernel

The SDK project must be upgraded with the newly generated hardware description file in the previous section. Before using the custom IPs in our application, we are first required to initialize them as follows -

```
#include "xparameters.h"

// Instance of the Custom IP
XConv_50x2 xconv;
XConv_50x2_Config *xconv_cfg;

XConv_100x4 xconvL;
XConv_100x4_Config *xconvL_cfg;

xconv_cfg = XConv_50x2_LookupConfig(XPAR_CONV_50X2_0_DEVICE_ID);
if(xconv_cfg)
{
        status = XConv_50x2_CfgInitialize(&xconv, xconv_cfg);
        if(status != XST_SUCCESS)
                return XST_FAILURE;
}

xconvL_cfg = XConv_100x4_LookupConfig(XPAR_CONV_100X4_0_DEVICE_ID);
if(xconvL_cfg)
{
        status = XConv_100x4_CfgInitialize(&xconvL, xconvL_cfg);
        if(status != XST_SUCCESS)
                return XST_FAILURE;
}
```

Next, we will make changes in the TFLite kernel for convolution operation (float) which is located at -
<sdk project >/tensorflow_lite/tensorflow/lite/kernels/internal/reference/**conv.h**.

By default, the kernel code uses normal for loops for computation as follows -

```
for (int batch = 0; batch < batches; ++batch) {
    for (int out_y = 0; out_y < output_height; ++out_y) {
      const int in_y_origin = (out_y * stride_height) - pad_height;
      for (int out_x = 0; out_x < output_width; ++out_x) {
        const int in_x_origin = (out_x * stride_width) - pad_width;
        for (int out_channel = 0; out_channel < output_depth; ++out_channel) {

          float total = 0.f;
          for (int filter_y = 0; filter_y < filter_height; ++filter_y) {
            const int in_y = in_y_origin + dilation_height_factor * filter_y;
            for (int filter_x = 0; filter_x < filter_width; ++filter_x) {
              const int in_x = in_x_origin + dilation_width_factor * filter_x;

              // Zero padding by omitting the areas outside the image.
              const bool is_point_inside_image =
                  (in_x >= 0) && (in_x < input_width) && (in_y >= 0) &&
```

```
                (in_y < input_height);

            if (!is_point_inside_image) {
              continue;
            }
            for (int in_channel = 0; in_channel < input_depth; ++in_channel) {
              float input_value = input_data[Offset(input_shape, batch, in_y,
                                                    in_x, in_channel)];
              float filter_value = filter_data[Offset(
                  filter_shape, out_channel, filter_y, filter_x, in_channel)];
              total += (input_value * filter_value);
            }
          }
        }
        float bias_value = 0.0f;
        if (bias_data) {
          bias_value = bias_data[out_channel];
        }
        output_data[Offset(output_shape, batch, out_y, out_x, out_channel)] =
            ActivationFunctionWithMinMax(total + bias_value,
                                         output_activation_min,
                                         output_activation_max);
      }
    }
  }
}
```

The value of variable *batches* is generally 1. Thus, we can ignore this variable for simplifying the loop. Instead of iterating over each element, we can perform the above operation using our custom IPs as follows -

```
if(input_height==100)     // CNN Layer with input dimension 100x4
{
  float resL[400];   // Temporary variable to hold the 2D convolution output
  // Set Inputs
  XConv_100x4_Set_depth(&xconvL, input_depth);
  XConv_100x4_Set_out_channels(&xconvL, output_depth);
  XConv_100x4_Set_im(&xconvL, (u32) input_data);
  XConv_100x4_Set_kernel(&xconvL, (u32) filter_data);
  XConv_100x4_Set_bias(&xconvL, (u32) bias_data);
  XConv_100x4_Set_result(&xconvL, (u32) resL);

  for (int out_channel = 0; out_channel < output_depth; ++out_channel)
  {
          XConv_100x4_Set_curr_channel(&xconvL, out_channel);

          for (int in_channel = 0; in_channel < input_depth; ++in_channel)
          {
                  XConv_100x4_Set_curr_depth(&xconvL, in_channel);

                  // Invalidate the data cache variable for coherency
                  microblaze_invalidate_dcache_range((u32)resL, 1600);
```

```
                XConv_100x4_Start(&xconvL);  // Start the computation
                while(!XConv_100x4_IsDone(&xconvL));  // Wait till over

        // 3D convolution - Iterative accumulation
                for(int j=0; j<100; j++)
                        for(int i=0; i<4; i++)
                        {
                                int idx1 = i + j*4;
                                int idx2 = idx1*output_depth + out_channel;
                                if(in_channel==0)
                                        output_data[idx2] = resL[idx1];
                                else
                                        output_data[idx2] += resL[idx1];
                        }
        }
  }
}

else  // CNN Layer with input dimension 50x2
{
  float res[100];   // Temporary variable to hold the 2D convolution output
  // Set Inputs
  XConv_50x2_Set_depth(&xconv, input_depth);
  XConv_50x2_Set_out_channels(&xconv, output_depth);
  XConv_50x2_Set_im(&xconv, (u32) input_data);
  XConv_50x2_Set_kernel(&xconv, (u32) filter_data);
  XConv_50x2_Set_bias(&xconv, (u32) bias_data);
  XConv_50x2_Set_result(&xconv, (u32) res);

  for (int out_channel = 0; out_channel < output_depth; ++out_channel)
  {
        XConv_50x2_Set_curr_channel(&xconv, out_channel);

        for (int in_channel = 0; in_channel < input_depth; ++in_channel)
        {
        XConv_50x2_Set_curr_depth(&xconv, in_channel);

        // Invalidate the data cache variable for coherency
                microblaze_invalidate_dcache_range((u32)res, 400);

                XConv_50x2_Start(&xconv);     // Start the computation
                while(!XConv_50x2_IsDone(&xconv));    // Wait till over

                // 3D convolution - Iterative accumulation
                for(int j=0; j<50; j++)
                        for(int i=0; i<2; i++)
                        {
                                int idx1 = i + j*2;
                                int idx2 = idx1*output_depth + out_channel;
                                if(in_channel==0)
                                        output_data[idx2] = res[idx1];
                                else
                                        output_data[idx2] += res[idx1];
                        }
```

```
        }
  }
}

// Apply activation function
for (int out_y = 0; out_y < output_height; ++out_y) {
  for (int out_x = 0; out_x < output_width; ++out_x) {
        for (int out_channel = 0; out_channel < output_depth; ++out_channel) {
              int idx = Offset(output_shape, 0, out_y, out_x, out_channel);
              float val_out = output_data[idx];
                    output_data[idx] = ActivationFunctionWithMinMax(val_out,
                                output_activation_min,
                                output_activation_max);
        }
  }
}
```

Notice that we invalidate the data cache for our temporary variable after performing each 2D convolution. This is necessary to maintain the microblaze cache coherency with the external DDR3 memory. Also, the 3D convolution is performed by iteratively invoking the CNN IP and accumulating the partial results. Next, we can proceed to run and benchmark our application.

The new inference time for predicting a single action on the Arty FPGA board is **1.00 s** @100MHz. For some pair of sequence having length of 1500 each, the total alignment time is $\approx$ 25 min. Thus, we have achieved an acceleration of about **2X** using our custom accelerator IPs, **regardless** of using a slower external memory (DDR3) and working with floating precision! We have also achieved better performance as compared to the ST Disco board implementation.

This chapter was centered around designing the CNN hardware accelerators and further integrating them with the MicroBlaze design. These IPs can drastically reduce the application run-time. We can create our own DNN accelerator chips for various ASIC applications. A future work could be to implement a more general design which offers the flexibility for changing various model parameters during the run-time. Such a design would be highly advantageous due to its portability and performance gains.

Chapter 9

Conclusion & Future Work

9.1 Conclusion

We have developed and demonstrated the toolchain for deploying the machine learning applications on edge devices, including the resource-constrained FPGAs. We have also demonstrated a Deep Reinforcement learning-based Genomics application of DNA Pairwise sequence alignment on the edge devices, along with the design and implementation of the CNN hardware accelerators for improving our application inference timings ($\approx$ 2X). A detailed performance analysis and benchmarking on suitable datasets is also carried out in this project.

*Few **novelty** in our work include the following -

- **Technique for Model size reduction using AutoML**: Although AutoML is normally used as a tool for training the deep neural networks without any model description, this is the first time that it has been used for reducing the size of a pre-trained model. Also, AutoML is more relevant for supervised learning problems, however, we were able to use this in a reinforcement learning setup. The final results show an order of reduction in the model parameters with only slight performance degradation in few cases.

- **Usage of BLAST alignment score for training and benchmarking**: We referred a paper by Song et al. for understanding how Deep RL can be employed in a sequence alignment task [18]. However, they have analysed the performance of their model by using only the total number of obtained matches. We, on the other hand, are using a more concrete metric called BLAST alignment score [29] which quantifies how good our alingment is. This scoring system

also accounts for the effect of gaps and mis-matches in the generated alignment. Further, we have used this scoring system to decide our reward system during the RL agent training as well as for the benchmarking purposes on the Influenza dataset.

- **Design and implementation of CNN hardware accelerators using Vivado HLS**: It is true that a lot of research has already been done in the design and implementation of the CNN hardware accelerators. For example, the Eyeriss [41] architecture from MIT is also based on this row-stationary approach. In fact, we referred this architecture during our own implementation. However, we weren't able to discover any sources which layout the complete implementation of such accelerator and its further integration with the hardware design. In short, the above design(s) are not open-sourced and it's not clear which framework they have used for the implementation. We have understood the design and implemented it from scratch using the Vivado HLS toolchain. Such a design can be easily integrated with any other HLS solution and successive improvements in the performance can be made due to flexibility & easy debugging which the HLS tool offers.

- **Usage of DSP instructions for accelerating applications on Cortex-M series**: During our first phase, we were required to accelerate the inference timings of our TFLite applications which were running on the ST boards with Cortex-M cores. We used several DSP instructions which are provided by the ARM CMSIS library in order to perform this task. For example, in the fully-connected layer, instead of computing each element during the matrix-multiplication operation iteratively, we can use special DSP instructions which can directly multiply a vector with a matrix. The corresponding performance gain was also mentioned in the results section.

- **Deployment of Machine Learning frameworks on ARM Cortex-M3 soft IP & Xilinx MicroBlaze**: The TFLite framework is meant to be deployed on devices powered by microcontroller units. However, we were also able to run such applications on the resource-constrained FPGA devices. This required several optimizations in the hardware design such as designing the path for accessing the data memory (shared among IPs and MicroBlaze), performance improvements in MicroBlaze, etc. With an implementation on FPGA, we can create our own custom IPs to accelerate the application inference timings. This gives even

more flexibilty for deploying the machine learning applications. With such an implementation, we can even create our own custom ASIC chips.

*Our work would prove crucial for implementing various machine learning-based applications on edge devices and optimizing the obtained performance. This makes machine learning even more **accessible** and **affordable** than before. At the end, our future is contained in smaller and intelligent devices!

9.2 Future Work

This project can be further extended to work in the following areas -

- Improvement in the current application model: We might work towards improving the accuracy of our Deep RL agent. For ex, we can try training with different, yet innovative reward systems and model architecture which is both time and space efficient. The window size can also be made adaptable with the diversity in input DNA sequences.
- Designing a general CNN Hardware accelerator: The current CNN accelerator IP is hard-coded in terms of the input and kernel dimensions, stride value, etc. We can try synthesizing the design with adjustable scalar inputs (such as dimensions, padding, stride, etc.). Such a design would be highly portable and can be easily integrated with future applications. The usage of our in-house HDL toolchain such as **Ahir** (AA language) is also proposed for improving the throughput. Ahir can be very useful for exploiting the maximum available parallelism in the design.
- New application: We can definitely work towards a new machine learning-based application which needs to be deployed on a smaller device along with performing various hardware optimizations. Deep reinforcement learning is a powerful tool for solving various challenging problems including self-driving car, drone navigation, bio-science, hardware chatbots, etc.

This field carries huge potential for a lot of interesting innovations and improving our day-to-day life. The future will be brighter with these small hand-held companions. A true example of Machine-Human friendship!

9.3 Questions from External Examiners

Following were few of the questions raised by the external examiners during the evaluation of this project. Corresponding answers are also highlighted as follows -

- **Fixed point implementation for EdgeAlign?**: We are currently using the Tensorflow Lite for Microcontroller framework for deploying our machine learning models on the edge devices. Unfortunately, at present, this framework doesn't supports the fixed point computation [42]. We can definitely watch for future realeases and add the required functionality.

- **Applications of TinyML**: We have developed and demonstrated the toolchain for deploying various machine learning applications on the edge devices. We have also demonstrated that it is even possible to deploy few complex applications such as DNA sequence alignment with appropriate software and hardware optimizations. Such TinyML implementation has vast applications ranging from AgroTech, IoT, Healtcare, etc. For example, we can generate an object detection model and embed it in a suitable hardware device such as CCTVs etc. Structural health monitoring is yet another application where hardware resources can be deployed even in the remote areas. Few other applications might include automated drone navigation, hardware chatbots, etc.

Bibliography

[1] R. Sanchez-Iborra and A. F. Skarmeta, "Tinyml-enabled frugal smart objects: Challenges and opportunities," *IEEE Circuits and Systems Magazine*, vol. 20, no. 3, pp. 4–18, 2020.

[2] P. Crum, "Hearables: Here come the: Technology tucked inside your ears will augment your daily life," *IEEE Spectrum*, vol. 56, no. 5, pp. 38–43, 2019.

[3] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, *et al.*, "Tensorflow lite micro: Embedded machine learning on tinyml systems," *arXiv preprint arXiv:2010.08678*, 2020.

[4] *https://www.tensorflow.org/lite/microcontrollers/build_convert#model_conversion*.

[5] *https://www.tensorflow.org/lite/performance/post_training_quantization*.

[6] *https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html*.

[7] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *arXiv preprint arXiv:1801.06601*, 2018.

[8] *https://www.arm.com/resources/free-arm-cortex-m-on-fpga*.

[9] *https://www.xilinx.com/products/design-tools/microblaze.html*.

[10] *https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug984-vivado-microblaze-ref.pdf*.

[11] *https://www.ebi.ac.uk/Tools/psa/*.

[12] *https://blast.ncbi.nlm.nih.gov/Blast.cgi*.

[13] S. McGinnis and T. L. Madden, "Blast: at the core of a powerful and diverse set of sequence analysis tools," *Nucleic acids research*, vol. 32, no. suppl_2, pp. W20–W25, 2004.

[14] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[15] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*, pp. 1995–2003, PMLR, 2016.

[16] I.-G. MIRCEA, M.-I. BOCICOR, and A. DÎINCU, "On reinforcement learning based multiple sequence alignment.," *Studia Universitatis Babes-Bolyai, Informatica*, vol. 59, no. 2, 2014.

[17] I.-G. Mircea, I. Bocicor, and G. Czibula, "A reinforcement learning based approach to multiple sequence alignment," in *International Workshop Soft Computing Applications*, pp. 54–70, Springer, 2016.

[18] Y.-J. Song, D. J. Ji, H. Seo, G. B. Han, and D.-H. Cho, "Pairwise heuristic sequence alignment algorithm based on deep reinforcement learning," *IEEE Open Journal of Engineering in Medicine and Biology*, vol. 2, pp. 36–43, 2021.

[19] R. Jafari, M. M. Javidi, and M. Kuchaki Rafsanjani, "Using deep reinforcement learning approach for solving the multiple sequence alignment problem," *SN Applied Sciences*, vol. 1, no. 6, pp. 1–12, 2019.

[20] R. K. Ramakrishnan, J. Singh, and M. Blanchette, "Rlalign: a reinforcement learning approach for multiple sequence alignment," in *2018 IEEE 18th International Conference on Bioinformatics and Bioengineering (BIBE)*, pp. 61–66, IEEE, 2018.

[21] R. Joeres, "Multiple sequence alignment using deep reinforcement learning," *SKILL 2021*, 2021.

[22] R. C. Edgar, "Muscle: multiple sequence alignment with high accuracy and high throughput," *Nucleic acids research*, vol. 32, no. 5, pp. 1792–1797, 2004.

[23] K. Katoh and D. M. Standley, "Mafft multiple sequence alignment software version 7: improvements in performance and usability," *Molecular biology and evolution*, vol. 30, no. 4, pp. 772–780, 2013.

[24] F. Sievers, A. Wilm, D. Dineen, T. J. Gibson, K. Karplus, W. Li, R. Lopez, H. McWilliam, M. Remmert, J. Söding, *et al.*, "Fast, scalable generation of high-quality protein multiple sequence alignments using clustal omega," *Molecular systems biology*, vol. 7, no. 1, p. 539, 2011.

[25] M. A. Larkin, G. Blackshields, N. P. Brown, R. Chenna, P. A. McGettigan, H. McWilliam, F. Valentin, I. M. Wallace, A. Wilm, R. Lopez, *et al.*, "Clustal w and clustal x version 2.0," *bioinformatics*, vol. 23, no. 21, pp. 2947–2948, 2007.

[26] K.-M. Chao, W. R. Pearson, and W. Miller, "Aligning two sequences within a specified diagonal band," *Bioinformatics*, vol. 8, no. 5, pp. 481–487, 1992.

[27] C. Camacho, G. Coulouris, V. Avagyan, N. Ma, J. Papadopoulos, K. Bealer, and T. L. Madden, "Blast+: architecture and applications," *BMC bioinformatics*, vol. 10, no. 1, pp. 1–9, 2009.

[28] G. Marçais, A. L. Delcher, A. M. Phillippy, R. Coston, S. L. Salzberg, and A. Zimin, "Mummer4: A fast and versatile genome alignment system," *PLoS computational biology*, vol. 14, no. 1, p. e1005944, 2018.

[29] *https://www.arabidopsis.org/Blast/BLASToptions.jsp*.

[30] *https://www.ncbi.nlm.nih.gov/books/NBK1734/#:∼:text=A%20BLAST%20alignment%20consists%20of,the%20length%20of%20the%20alignment.*

[31] T. H. Jukes, C. R. Cantor, *et al.*, "Evolution of protein molecules," *Mammalian protein metabolism*, vol. 3, pp. 21–132, 1969.

[32] *https://www.mja.com.au/journal/2014/201/1/impact-genomics-future-medicine-and-health*.

[33] P. Mitra and P. Sharma, "Poct in developing countries," *EJIFCC*, vol. 32, no. 2, p. 195, 2021.

[34] *https://nanoporetech.com/applications/dna-nanopore-sequencing*.

[35] *https://nanoporetech.com/products/minion*.

[36] B. Qian and R. A. Goldstein, "Distribution of indel lengths," *Proteins: Structure, Function, and Bioinformatics*, vol. 45, no. 1, pp. 102–104, 2001.

[37] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1946–1956, ACM, 2019.

[38] M. Plappert, "keras-rl." https://github.com/keras-rl/keras-rl, 2016.

[39] *https://keras-rl.readthedocs.io/en/latest/agents/dqn/*.

[40] *https://www.ncbi.nlm.nih.gov/genomes/FLU/Database/nph-select.cgi*.

[41] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, pp. 262–263, 2016.

[42] *https://www.tensorflow.org/lite/api*$_d$*ocs/swift/Structs/Tensor/DataType*.