HLS for BrainTumor Detection Using MRI

Group Number: 17

Dhruvkumar Kakadiya (234101013)

Ajaypal Singh (234101005)

Jash Ratanghayra (234101019)

Kishan Thakkar (234101024)

Gautam Gandhi (234101014)

Que 1: Description of the model. Please write briefly about the ML model given to you along with the following data.

- **Tasks of Model:** Detect MRI of the brain and classify them into (brain with tumor and brain with no tumor).

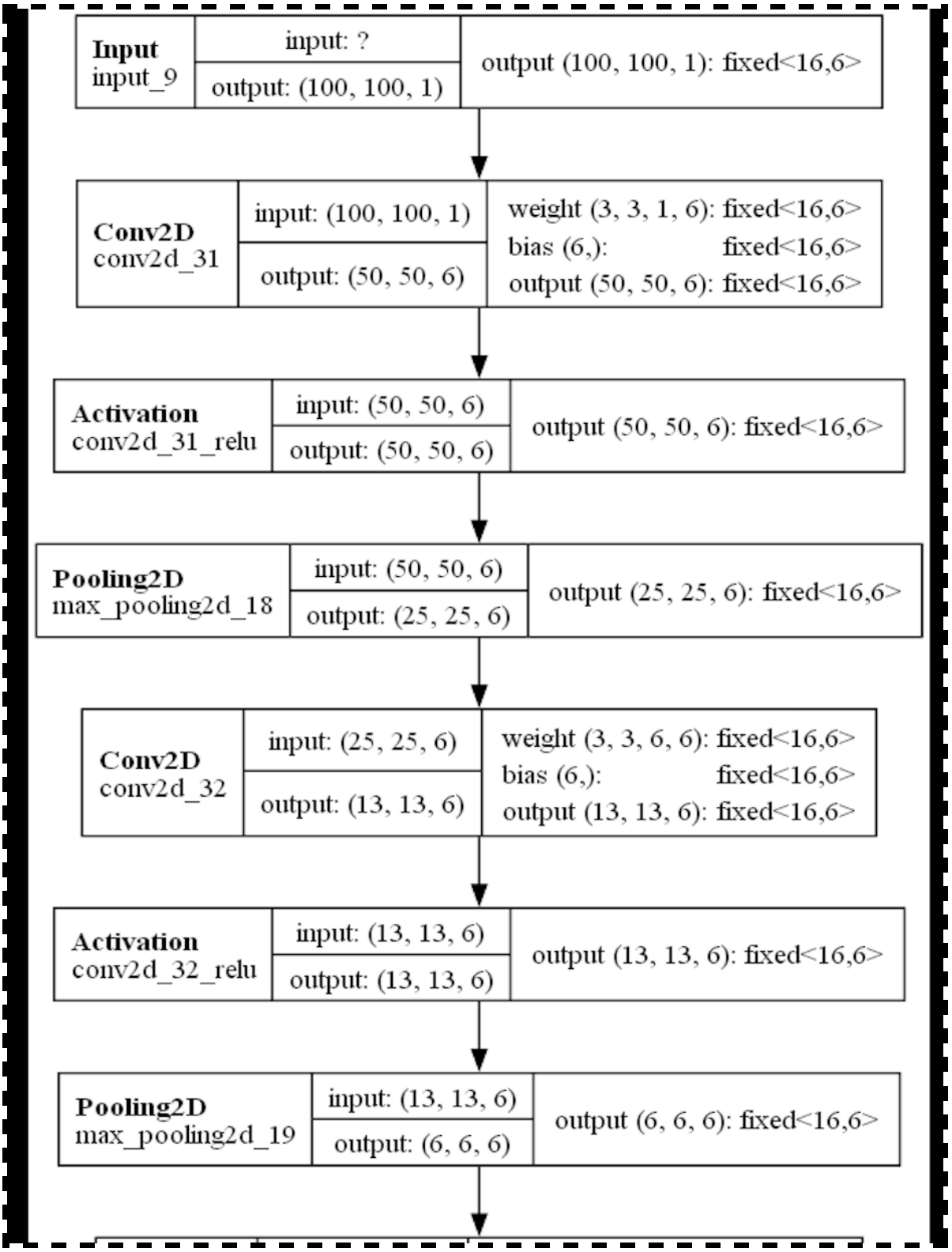
- **Number of Layers**: 14 layers in total.

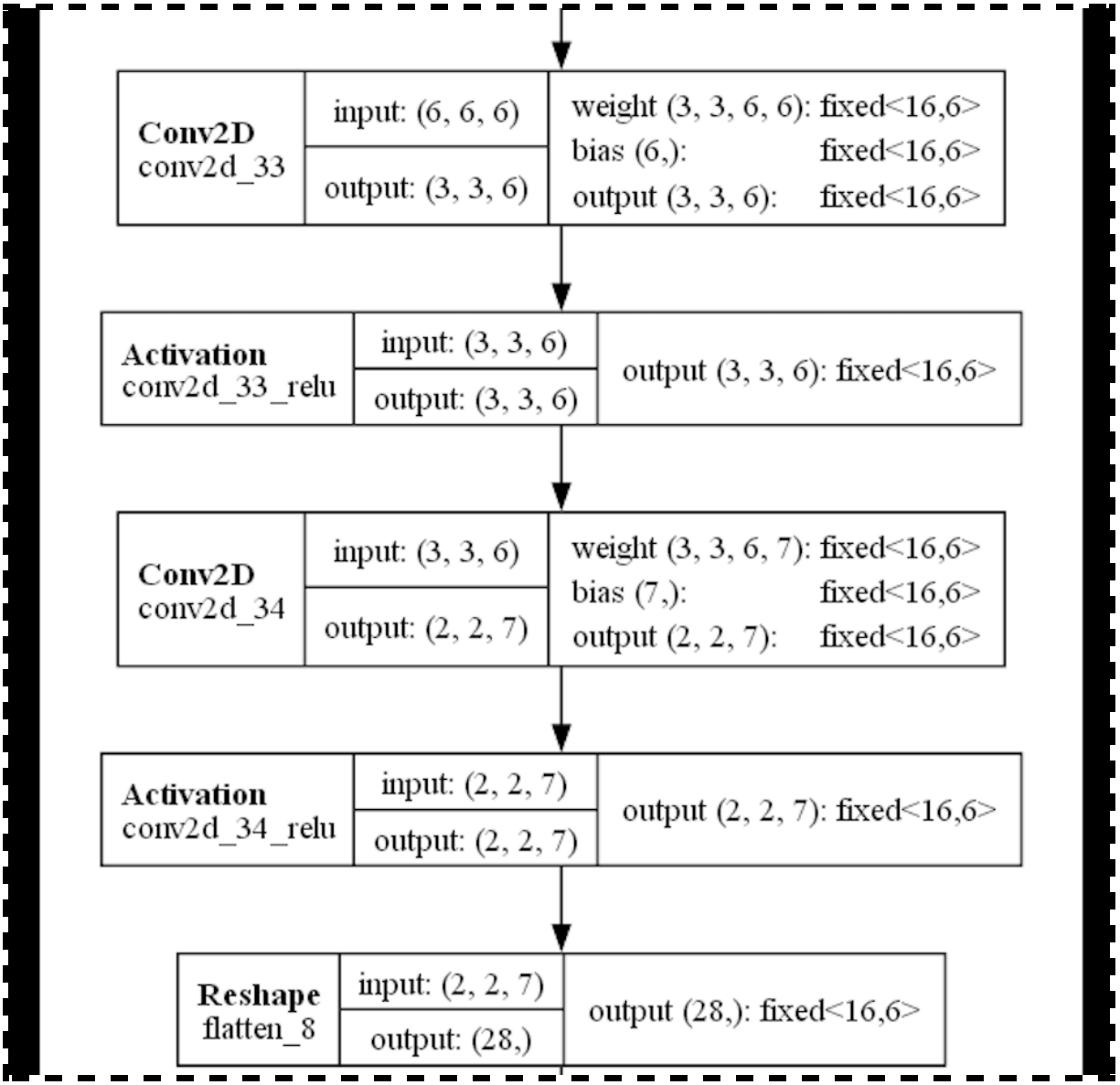
- **Type of Layers:**
    - Padding layer 2D.
    - Convolution layer 2D.
    - Max Pooling layer 2D.
    - Dense layer.
    - Flatten layer.

- **Other Details:**

The machine learning model used in this project employs Convolutional Neural Networks (CNNs) to detect brain tumors from MRI images. The model architecture consists of three main layers: convolutional, pooling, and fully connected. Convolutional layers extract features from the input MRI images,

capturing patterns that signify the presence of tumors. Pooling layers reduces computational complexity while preserving essential information. Finally, fully connected layers combine the learned features.





Many more layers…….

To enhance model performance, the model utilizes two popular activation functions: Tanh and ReLU. Tanh introduces non-linearity to the network, helping to model complex relationships within the data, while ReLU (Rectified Linear Unit) efficiently handles the vanishing gradient problem by allowing only positive values to pass through.

Que 2: Changes made to make keras2c generated files synthesizable and a brief description of the change made.

1. Original code:

```
memcpy(&output->array[offset], &input->array[i*num],
num*sizeof(input->array[0]));
```

Modified code:

```
for (size_t ji = 0; ji < num; ji++) {
        output->array[offset + ji] = input->array[i * num + ji];
    }
```

Reason:

'memcpy' is used in software to copy a block of memory from one location to another. In hardware, memory operations are typically handled differently. Memory in hardware is represented as registers, block RAM (BRAM), or other forms of physical memory storage, which have different access patterns and capabilities compared to a general-purpose CPU's memory architecture.

The modified code replaces memcpy() with a loop that manually copies each element from input->array to output->array. By iterating over each element individually, the code ensures that it is synthesizable because it directly translates into hardware logic that copies each element sequentially. This approach eliminates the need for handling different memory regions and unaligned data, making it suitable for synthesis in Vivado.

2. Original code:

```
memset(C, 0, outrows*outcols*sizeof(C[0]));
```

Modified code:

```
for (size_t row = 0; row < outrows; ++row) {
            // Iterate over each column of C
            for (size_t col = 0; col < outcols; ++col) {
               // Set each element of C to zero
                    C[row * outcols + col] = 0.0f;
            }
 }
```

Reason:

The reason for modifying the code is that using memset in Vivado is incompatible with its hardware-based memory management system, potentially causing unexpected behavior. Vivado requires special functions for memory manipulation instead of direct memory writes like memset.

The modified code replaces memset with a nested loop to iterate over each element of the array C and set them to zero individually. This approach ensures compatibility with Vivado's memory management system by avoiding direct memory writes. Additionally, it introduces a pragma directive #pragma HLS pipeline to potentially optimize the loop for FPGA synthesis, improving performance.

3. Original Code:

```
float dense_30_bias_array[70] = {...};
k2c_tensor dense_30_bias = {&dense_30_bias_array[0],1,70,{70, 1, 1, 1, 1}};
float dense_30_fwork[5964] = {0};
```

Modified code:

First, we declared the dense_30_bias object global in our C file.
Then the code is replaced by the below code

```
dense_30_bias.ndim=1;
dense_30_bias.numel=70;
dense_30_bias.shape[0]=1;
dense_30_bias.shape[1]=1;
dense_30_bias.shape[2]=1;
dense_30_bias.shape[3]=1;
dense_30_bias.shape[4]=1;
for(i=0;i<70;i++){
        dense_30_bias.array[i] = dense_30_bias_array[i];
}
```

```
1 error generated.\n
ERROR: [HLS 200-70] Compilation errors found: Pragma processor failed: In file included from vivado_test_1/test.c:1:
vivado_test_1/test.c:51:29: error: initializing 'float' with an expression of incompatible type 'float *'; remove &
k2c_tensor dense_30_bias = {&dense_30_bias_array[0],1,70,{70, 1, 1, 1, 1}};
                           ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 error generated.
```

Reason:
We encountered an error while trying to initialize the dense_30_bias structure. The error message indicates that we attempted to initialize a float with an expression of incompatible type float *. This occurred because we used the & operator, which generates a pointer while initializing the array member of the k2c_tensor structure, which should be of type float, not float *.

The modification directly assigns values to the members of the dense_30_bias structure without using the problematic initialization approach. Specifically, it sets the ndim, numel, and shape members of the dense_30_bias structure to the appropriate values and then loops through the dense_30_bias_array to assign its values to the array member of the dense_30_bias structure. This approach avoids the use of the & operator, resolving the type mismatch issue and making the code workable in HLS.

Defined Below functions in braintumor.c

- void k2c_relu_func(float * x, const size_t size)
- *void k2c_softmax_func(float* x, const size_t size)
- void k2c_matmul(float * C, const float * A, const float * B, const size_t outrows, const size_t outcols, const size_t innerdim)
- void k2c_idx2sub(const size_t idx, size_t * sub, const size_t * shape, const size_t ndim)
- size_t k2c_sub2idx(const size_t * sub, const size_t * shape, const size_t ndim)
- void k2c_pad2d(k2c_tensor* output, const k2c_tensor* input, const float fill, const size_t * pad)
- void k2c_dot(k2c_tensor* C, const k2c_tensor* A, const k2c_tensor* B, const size_t * axesA, const size_t * axesB, const size_t naxes, const int normalize, float * fwork)

- void k2c_conv2d(k2c_tensor* output, k2c_tensor* input, const k2c_tensor* kernel, const k2c_tensor* bias, const size_t* stride, const size_t* dilation)

- void k2c_maxpool2d(k2c_tensor* output, const k2c_tensor* input, const size_t * pool_size, const size_t * stride)

- void k2c_dense(k2c_tensor* output, const k2c_tensor* input, const k2c_tensor* kernel, const k2c_tensor* bias,int flag, float * fwork)

- void k2c_flatten(k2c_tensor *output, const k2c_tensor* input)

- void braintumer(k2c_tensor* input_9_input, k2c_tensor* dense_32_output)

Reason:

The above functions were originally defined across different header files. However, to streamline error resolution and enhance code maintainability, we consolidated their definitions into a single source file named braintumor.c. By centralizing these function definitions, we mitigate the need to navigate through various header files during debugging sessions. This consolidation not only simplifies the debugging process but also facilitates easier code management and reduces the risk of errors introduced by scattered function definitions across multiple files.

Change in k2c_tensor_include.h

Original code: float * array;

Modified code:
#define MAX_SIZE 16000
float array[MAX_SIZE];
Reason:

The original code declares a pointer to a floating-point array, which is not synthesizable in Vivado HLS due to hardware limitations. To address this, the

modified code replaces the pointer declaration with a fixed-size array using a #define directive, setting its maximum size to 16000 elements. By using a fixed-size array instead of a pointer, the modified code ensures compatibility with Vivado HLS's synthesis flow, allowing for static memory allocation and enabling successful hardware implementation.

Que 3: Changes made to generate HLS4ML report if a pragma is removed in this process. For each of the removed pragmas, a valid argument must be mentioned.

- **Tensorflow version:**
  The latest version of TensorFlow is not compatible with the hls4ml python library, the latest version is >= 2.16. But we required the stable version of TensorFlow "2.14.0". We need to install it separately.

```
166  set_part $part
167  # config_schedule -enable_dsp_full_reg=false
168  create_clock -period $clock_period -name default
```

- **enable_dsp_full_reg:** This setting, when enabled (true), likely configures the tool to fully register DSP blocks. DSP blocks (Digital Signal Processing) are specialized hardware units on FPGAs used for high-speed arithmetic operations, such as those found in signal processing and deep learning computations.

- While building the project we comment out this line because If the default behavior of the HLS tool is preferable for a specific project or part of a project, We might comment out this directive. Perhaps the default settings are more suitable for the specific design constraints or goals.

```
config['Model']['Precision'] = 'fixed<6,2>'
```

```
HLSConfig:
  LayerName:
    conv2d_31:
      Precision:
        bias: fixed<16,6>
        result: fixed<16,6>
        weight: fixed<16,6>
      Trace: false
    conv2d_31_relu:
      Precision:
        result: fixed<16,6>
      Trace: false
```

- WARNING: [ANALYSIS 214-1] Tool encounters 11760 load/store instructions to analyze which may result in long runtime.
  ERROR: [XFORM 203-1403] Unsupported enormous number of load/store instructions: 'nnet::dense_wrapper<ap_fixed<16, 6, (ap_q_mode)5, (ap_o_mode)3, 0>, ap_fixed<16, 6, (ap_q_mode)5, (ap_o_mode)3, 0>, config15>'.
  ERROR: [HLS 200-70] Failed building synthesis data model.
- Here we mentioned the fixed point data type for our model.
- 6: This is the total number of bits used for the number. This includes both the bits before and after the decimal point.
- 2: This indicates the number of bits that are used for the fractional part of the number (i.e., the digits after the decimal point).
- If we do not specify this then the range becomes out of bound.

```
// Use a function_instantiate in case it helps to explicitly optimize unchanging weights/biases
// //#pragma HLS function_instantiate variable=weights,biases
```

- For ‘nnet_dense_latency.h’.
- When used, this pragma can lead to increased resource utilization, as separate hardware might be allocated for handling operations on these variables each time they are referenced in a distinct function or context. This can be beneficial if it reduces contention or improves parallelism but can also consume more FPGA resources.
- If we do not comment on this pragma then we face the error like we have more weights and bias so we do not have enough FPGA resources for the same.

```
// //#pragma HLS ARRAY_PARTITION variable=weights block factor=2 // remove
//#pragma HLS ARRAY_RESHAPE variable=weights cyclic factor=2 dim=1
// //#pragma HLS ARRAY_PARTITION variable=biases complete
// //#pragma HLS ARRAY_PARTITION variable=mult complete
// //#pragma HLS ARRAY_PARTITION variable=acc complete
```

- For ‘nnet_dense_latency.h’.
- INFO: [XFORM 203-101] Partitioning array 'b6.V' in dimension 1 completely. ERROR: [XFORM 203-103] Array 'multi. V' (firmware/nnet

utils/nnet_dense_latency.h: 17): partitioned element 544) has exceeded the threshold (4096), which may cause long run-time. ERROR: [HLS 200-70] Pre-synthesis failed.

- For this reason here we remove this pragma of array partition.
- Here, we mentioned the error regarding only the 'mult.V' but we got the same error for biases and acc too.
- We remove all the pragma of partition of this array whenever we get this pragma with this variable.

```
data_T data_buf[CONFIG_T::n_pixels][mult_n_in];
//#pragma HLS ARRAY_PARTITION variable=data_buf complete dim=0

typename CONFIG_T::accum_t mult[mult_n_in * mult_n_out];
//#pragma HLS ARRAY_PARTITION variable=mult complete

typename CONFIG_T::accum_t acc[mult_n_out];
//#pragma HLS ARRAY_PARTITION variable=acc complete

//#pragma HLS ARRAY_PARTITION variable=weights complete
//#pragma HLS ARRAY_PARTITION variable=biases complete
```

- For 'nnet_conv2d_latency.h'.
- We remove this pragma also For the same reason that We mentioned above for the partition problem.

```
// Limit multipliers to control parallelization
//#pragma HLS ALLOCATION operation instances=mul limit=CONFIG_T::mult_config::multiplier_limit
```

- For 'nnet_conv2d_latency.h'.
- This directive is used to control the number of hardware resources (like DSP blocks) dedicated to multiplication operations in the synthesized design. By setting a limit, it helps manage the trade-off between performance (speed of operations and parallelism) and resource utilization (how much of the FPGAs available resources are used).
- It restricted the resource usage for particular operations, and the operation can not get their required FPGAs so for that reason we removed this pragma.

```
        Result:
            for (int i_res = 0; i_res < mult_n_out; i_res++) {
                //#pragma HLS UNROLL
                *(res++) = cast<data_T, res_T, typename CONFIG_T::mult_config>(acc[i_res]);
            }
        }
    }
```

- For ‘nnet_conv2d_latency.h’.
- In this case the pragma HLS UNROLL fully unrolled the loop that why it needed more resources for that reason we get error.
- Sometimes fully unrolled takes 1 cycle for iteration but this is not feasible because in one clock we have limitations.

Que 4: Markdown file for dependencies and version is given separately in zip file.

Q.5 Optimizations for each optimization applied (pragma), justify why it has been used.

1. **Unrolling Loops:**

Syntax: " #pragma HLS unroll "

We do loop unrolling when the number of iterations of the loop is fixed and the loop body is small which can be parallelized.

Purpose: Loop unrolling is a technique used to reduce loop overhead and improve throughput by replicating loop bodies multiple times.

Impact on Synthesis:

- **Latency:** Unrolling loops reduces the initiation interval, which is the number of clock cycles between consecutive iterations of a loop. This reduces latency, especially for tightly nested loops.

- **Resource Utilization**: Unrolling loops increases the number of hardware resources required, such as registers and logic elements, since each iteration of the loop is essentially replicated.

2. **Loop Pipelining:**

Syntax: " #pragma HLS pipeline "

If there is no resource constraint and no loop dependency then we can use a loop pipelining to improve throughput and latency but resource utilization will be high.

Purpose: Pipelining breaks down the operations within a loop into stages and executes them concurrently, allowing for higher throughput and reduced latency.

Impact on Synthesis:

- **Latency:** Pipelining can significantly reduce latency by allowing multiple loop iterations to be processed simultaneously.

- **Resource Utilization:** Pipelining increases resource utilization, particularly in terms of flip-flops and pipeline registers. However, it may also increase clock frequency by enabling better timing closure.

3. Dataflow Synthesis:

Syntax: " #pragma HLS dataflow "

This is an optimization technique used when we can pass or transfer the data between different modules of the program.

Purpose: Dataflow synthesis optimizes the hardware design by identifying independent operations and executing them concurrently, exploiting parallelism in the design.

Impact on Synthesis:

- **Latency:** Dataflow synthesis can reduce latency by maximizing parallelism in the design, allowing operations to proceed independently.

- **Resource Utilization:** It may increase resource utilization, particularly in terms of registers and routing resources, to enable concurrent execution of operations.

4. Partial Loop Unrolling:

Syntax: " #pragma HLS unroll factor=<factor> "

When the whole loop can not be unrolled at once we unroll the loop with a factor based on dependency in the loop content.

Purpose: Partial loop unrolling combines the benefits of loop unrolling and loop peeling. It unrolls only a portion of the loop, typically the first few iterations, to reduce loop overhead while maintaining flexibility.

Impact on Synthesis:

- **Latency:** Partial loop unrolling can reduce latency by reducing loop overhead for the initial iterations, which are often critical for performance.

- **Resource Utilization**: It increases resource utilization compared to full loop unrolling but may be more efficient in terms of resource allocation for critical loop segments.

Que 6: Results

- HLS4ML generated Latency and area overhead table.

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+---------+---------+------+
|       Name      | BRAM_18K| DSP48E|    FF   |   LUT   | URAM |
+-----------------+---------+-------+---------+---------+------+
|DSP              |        -|      -|        -|        -|     -|
|Expression       |        -|      -|        0|        2|     -|
|FIFO             |       95|      -|     4831|    14743|     -|
|Instance         |       23|      8|    13745|    65657|     -|
|Memory           |        -|      -|        -|        -|     -|
|Multiplexer      |        -|      -|        -|        -|     -|
|Register         |        -|      -|        -|        -|     -|
+-----------------+---------+-------+---------+---------+------+
|Total            |      118|      8|    18576|    80402|     0|
+-----------------+---------+-------+---------+---------+------+
|Available        |     5376|  12288|  3456000|  1728000|  1280|
+-----------------+---------+-------+---------+---------+------+
|Utilization (%)  |        2|   ~0  |    ~0   |        4|     0|
+-----------------+---------+-------+---------+---------+------+
```

```
+ Latency (clock cycles):
    * Summary:
    +--------+---------+--------+---------+----------+
    |      Latency      |     Interval      | Pipeline |
    |   min  |   max    |   min  |   max    |   Type   |
    +--------+---------+--------+---------+----------+
    |  367441|  3458344|  367440|  3458343| dataflow |
    +--------+---------+--------+---------+----------+
```

- Latency and area overhead for Baseline (Unoptimized).

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	2512289	2512289	2512289	2512290	2512290	2512290

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	661
FIFO	-	-	-	-
Instance	-	181	16648	19600
Memory	862	-	1184	58
Multiplexer	-	-	-	4306
Register	-	-	488	-
Total	862	181	18320	24625
Available	2360	2880	692800	346400
Utilization (%)	36	6	2	7

- Latency and area overhead table for Optimized (if there are multiple versions like various tradeoffs, give all of them).

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	6056
FIFO	-	-	-	-
Instance	0	238	28154	63912
Memory	858	-	896	35
Multiplexer	-	-	-	28622
Register	-	-	3997	-
Total	858	238	33047	98625
Available	2360	2880	692800	346400
Utilization (%)	36	8	4	28

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	1249664	1249664	1249664	1249665	1249665	1249665

- Finally, a comparison report of both optimized and HLS4ML generated reports.
- Provide the HLS results for each of the above scenarios in the below table.

Design	Normal	HLS4ML	Manual (Vivado)
LUT	24625	80402	98625
FF	18320	18576	33047
DSP	181	8	238
BRAM	862	118	858
Latency (min / max)	2512289 / same	367441 / 3458344	1249664 / same
Clock period	8.691 / 10	5.435 / 10	10.882 / 10