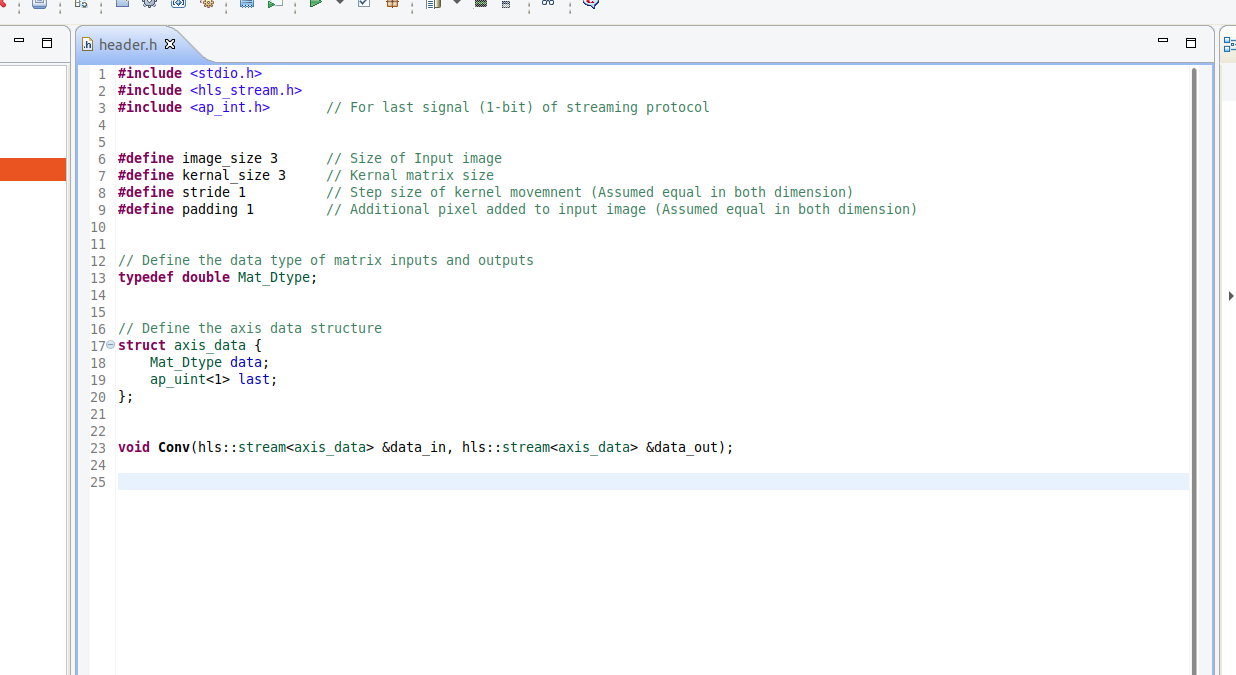
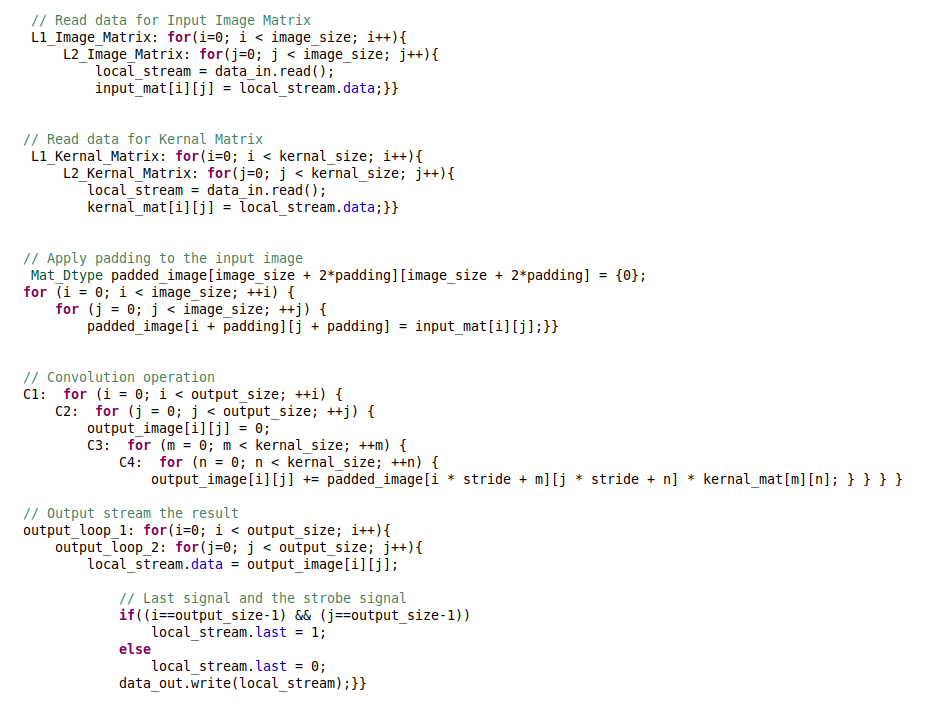
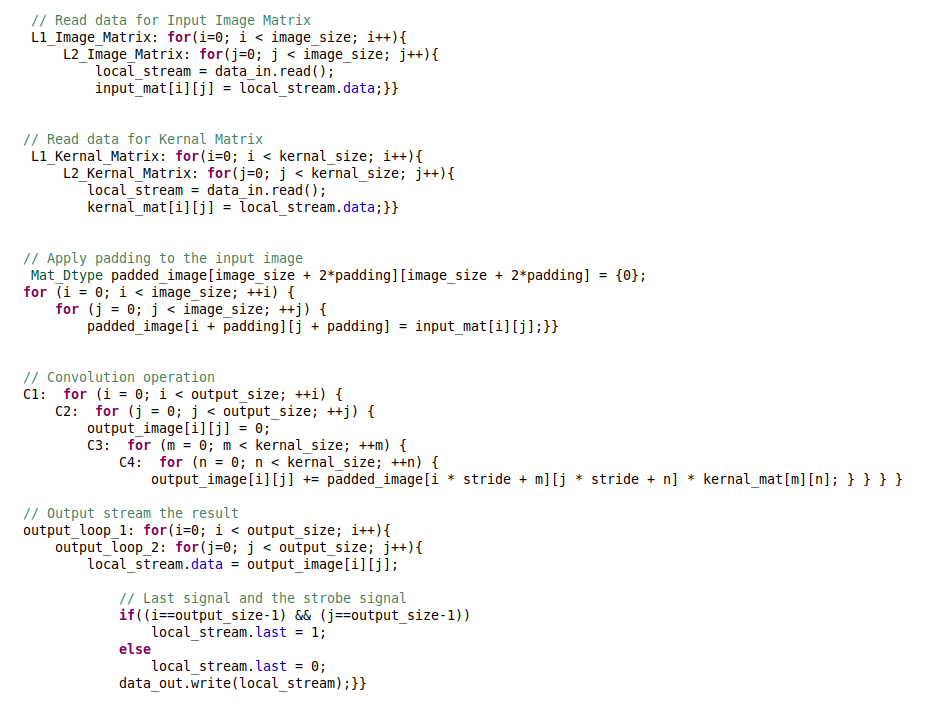
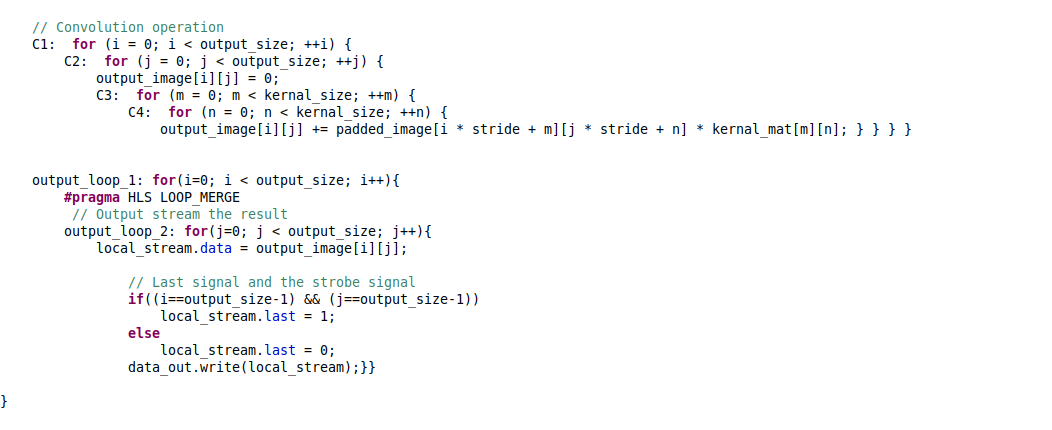
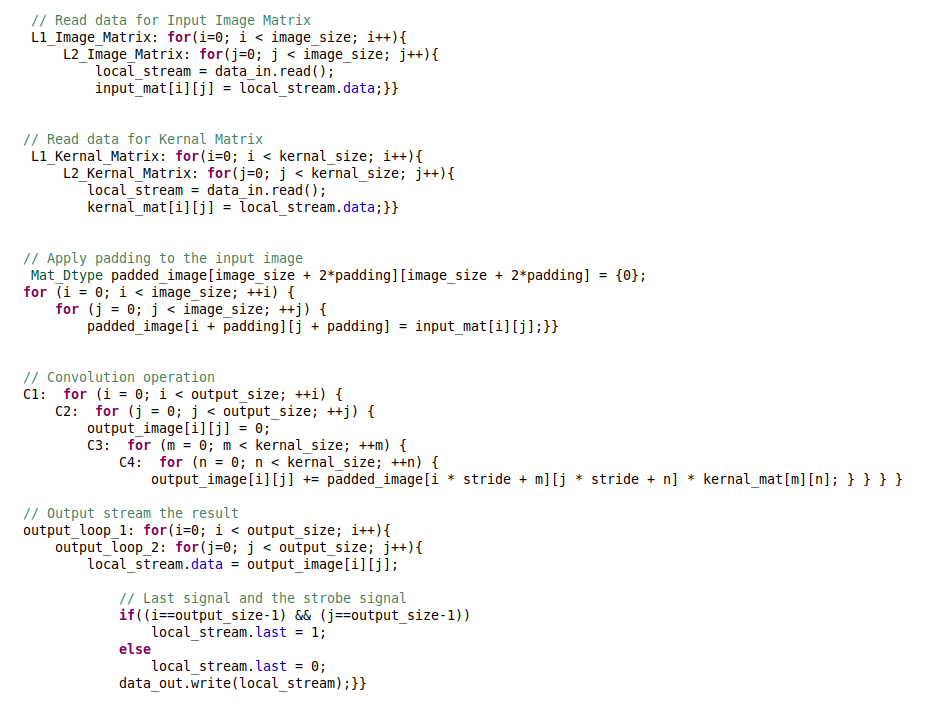
**Task description** - Implement a Conv2D operator and then utilize it to implement a Convolutional layer. The implementation should be tested on an Ultra96 board using a test dataset. Furthermore, precision scaling would be considered to explore accuracy-performance trade-offs. Both HDL and HLS can be used for the implementation.

**VIVADO HLS** - For this task I used custom IP, which performs convolution operation and has parameters like Image and kernel matrix size, stride and padding using Vivado HLS software. I choose HLS as it is easy and convenient to compile, simulate, debug and optimise algorithm implement in C/C++. The parameters are fixed but can be changed in the header file according to the requirements and again export the IP.

* The data is read and sent by IP using AXI input stream and AXI output stream channels create using below pragmas.
* #pragma HLS INTERFACE ap\_ctrl\_none port=return
* #pragma HLS INTERFACE axis register both port=data\_in
* #pragma HLS INTERFACE axis register both port=data\_out
* Image and kernel matrix are created and fed in values using stream input. Then padding is applied to the input image matrix. Then the convolution operation is carried out as below and then output image is sent via stream output channel.

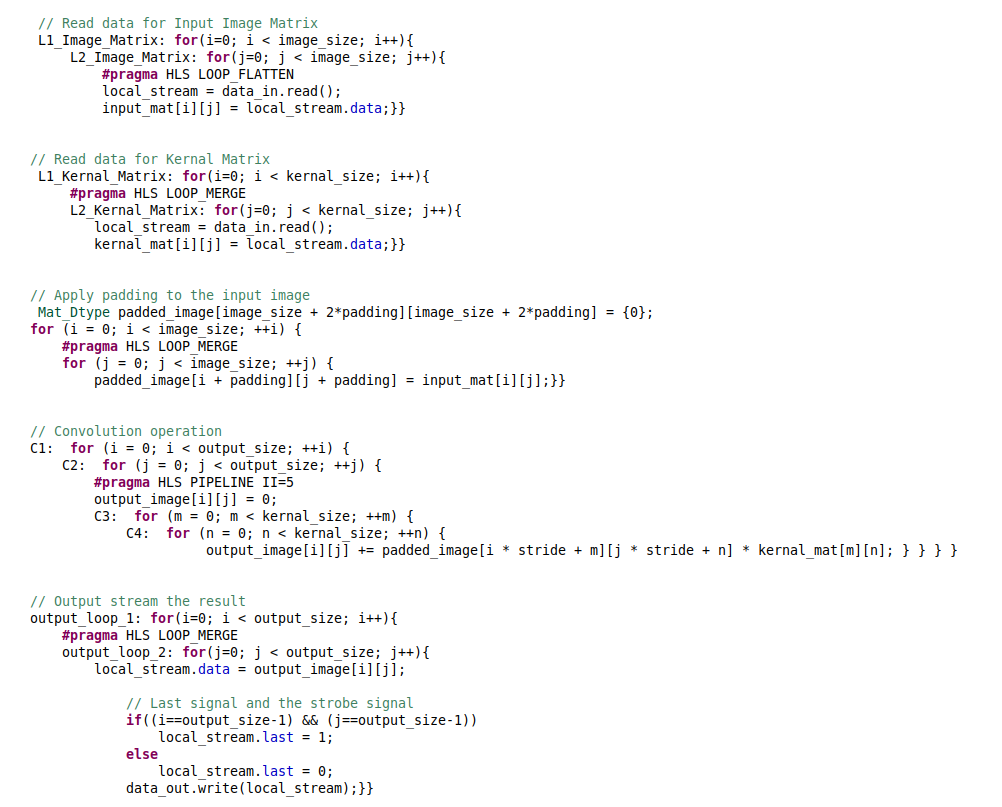
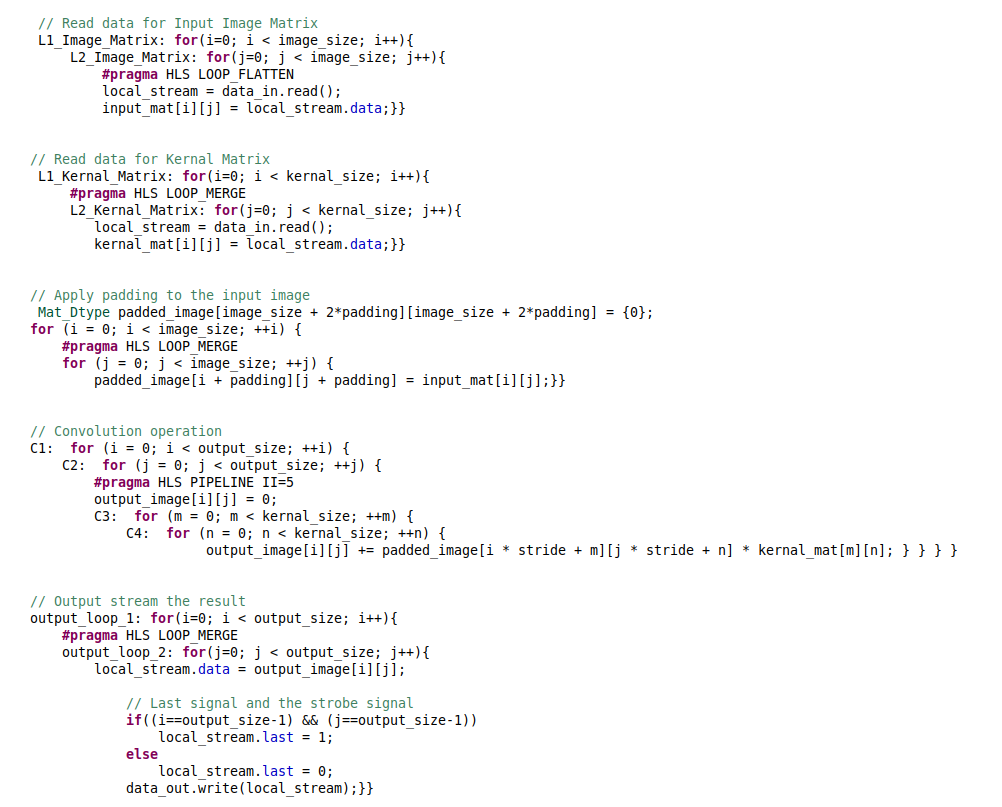


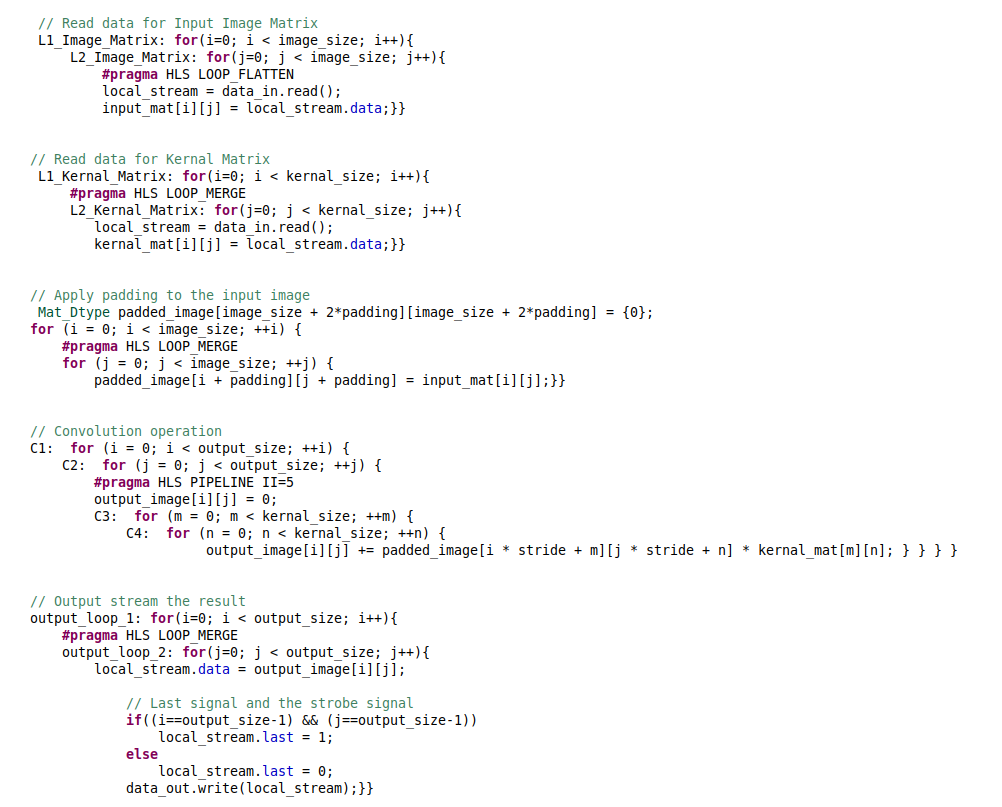




**Un-optimized code**

* 2 different types of IP were implemented and their performance were compared. One was without any optimisations using directives and other was fully optimised. For optimisation, I used following pragmas.
* **#pragma HLS LOOP\_FLATTEN** - Allows nested loops to be flattened into a single loop hierarchy with improved latency. This saves clock cycles, potentially allowing for greater optimization of the loop body logic.
* **#pragma HLS LOOP\_MERGE** - It merges consecutive loops into a single loop to reduce overall latency, increase sharing, and improve logic optimization. Reduces the number of clock cycles required in the RTL to transition between the loop-body implementations.
* **#pragma HLS PIPELINE II=5** – A pipelined function or loop can process new inputs every 5 clock cycles, where 5 is the II (initiation interval) of the loop or function. It allows the concurrent execution of operations.





**Optimized code**

* Analysing both design after synthesis

Table -1 Comparison between both design

|  |  |
| --- | --- |
| **Not – Optimised design** | **Optimised design** |
|  |  |
|  |  |
|  |  |

**VIVADO Design suite** - For this task I used Xilinx AXI DMA and controlled it from PYNQ using Jupiter notebook. DMA allows to stream data from memory to an AXI stream interface. This is called READ channel of the DMA. It can also read data from an AXI stream and write it back to memory using WRITE channel.

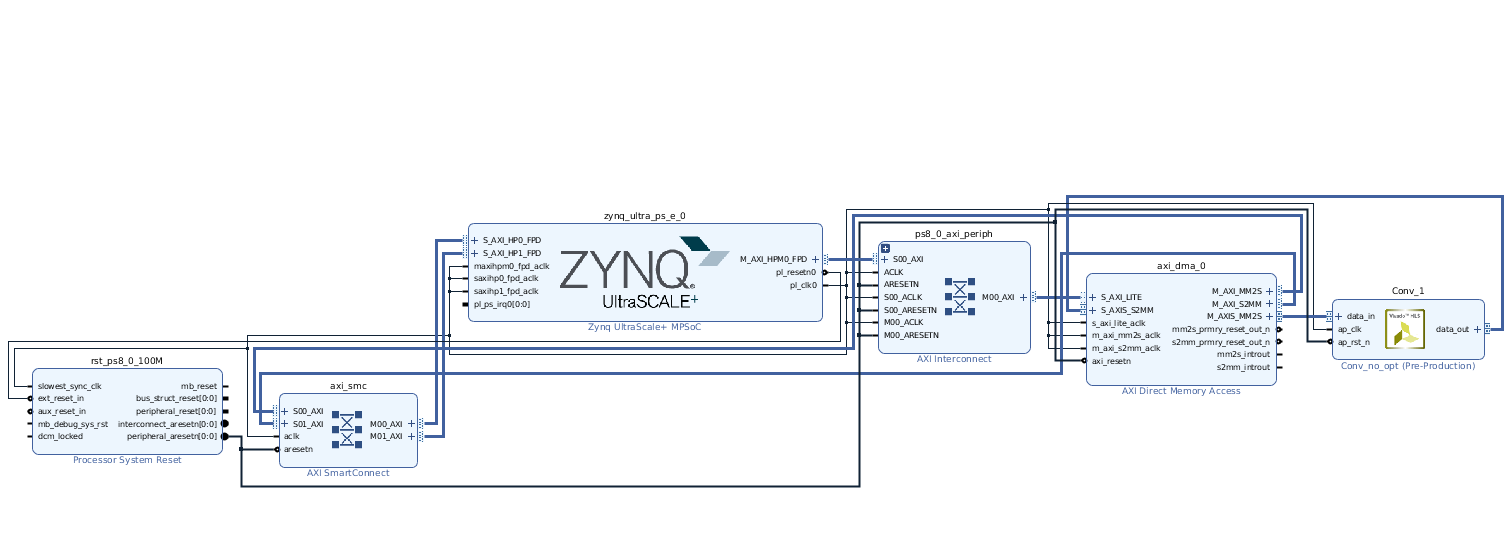
* This DMA is connected to Zynq Processing Subsystem (PS) with more peripherals like AXI interconnect, AXI smart connect and processor system reset.
* Configuring AXI DMA IP as follows.
* Width of buffer length register – 26 bits *(determines the maximum packet size for a single DMA transfer).*
* Address width – 64 bits *(because we are using Zynq Ultrascale+).*
* Read and write channels enabled
* Memory map data width - 64 bits *(defined in the PYNQ image and applied at boot time).*
* Stream data width – 64 bits *(match with IP stream width,* *for best results - match the settings of the DMA to your datapath).*
* After adding the IP to our Vivado block design is completed as below.

Figure-1 Vivado block design

* Creating HDL wrapper and generating bitstream files. Once completed repeated the same procedure for IP with optimised code.

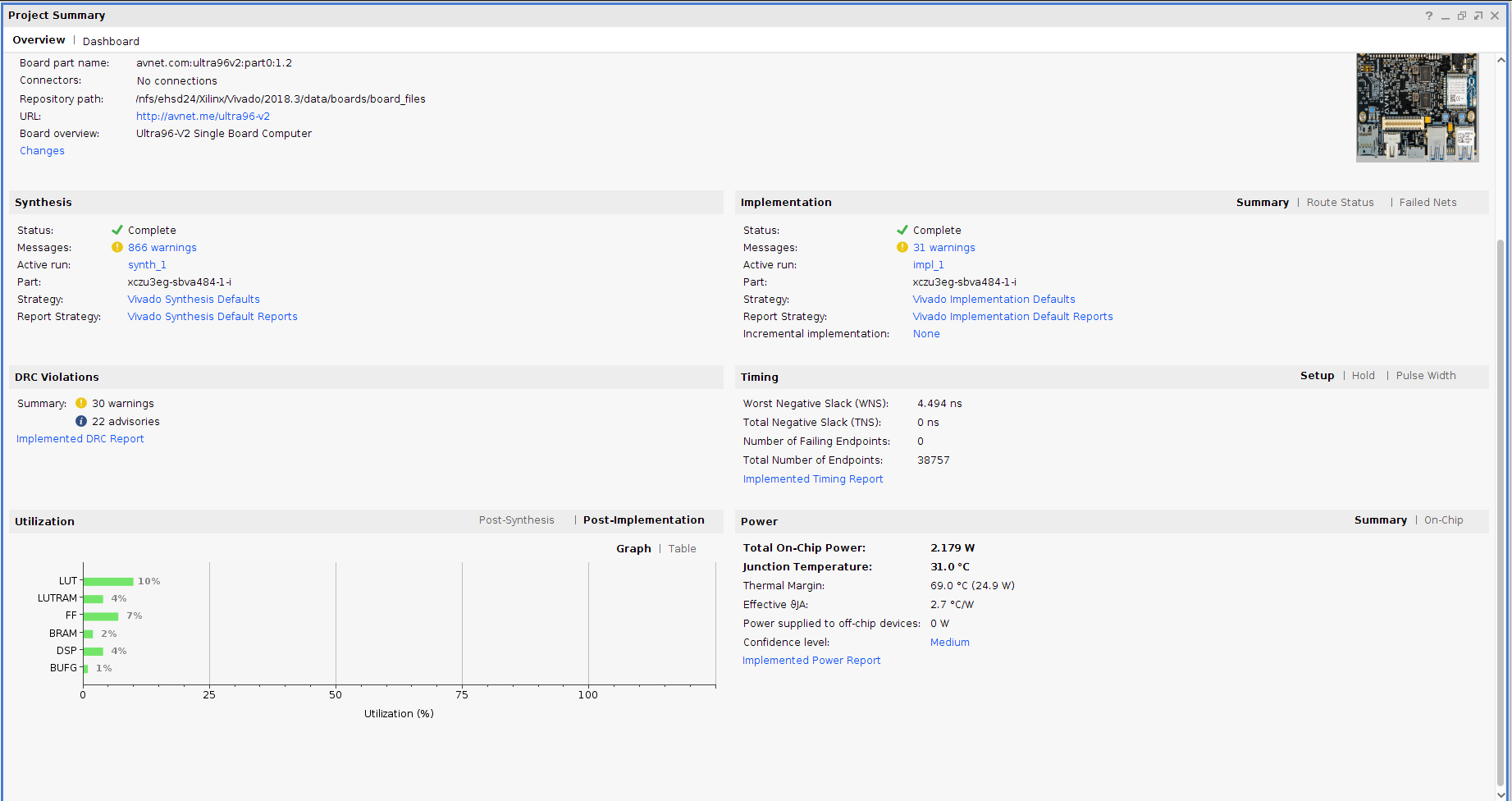
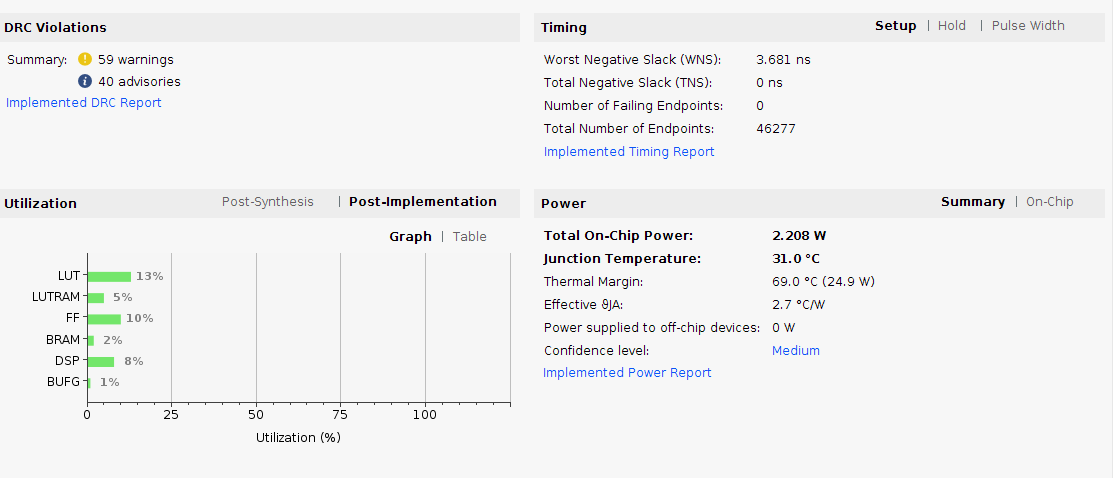
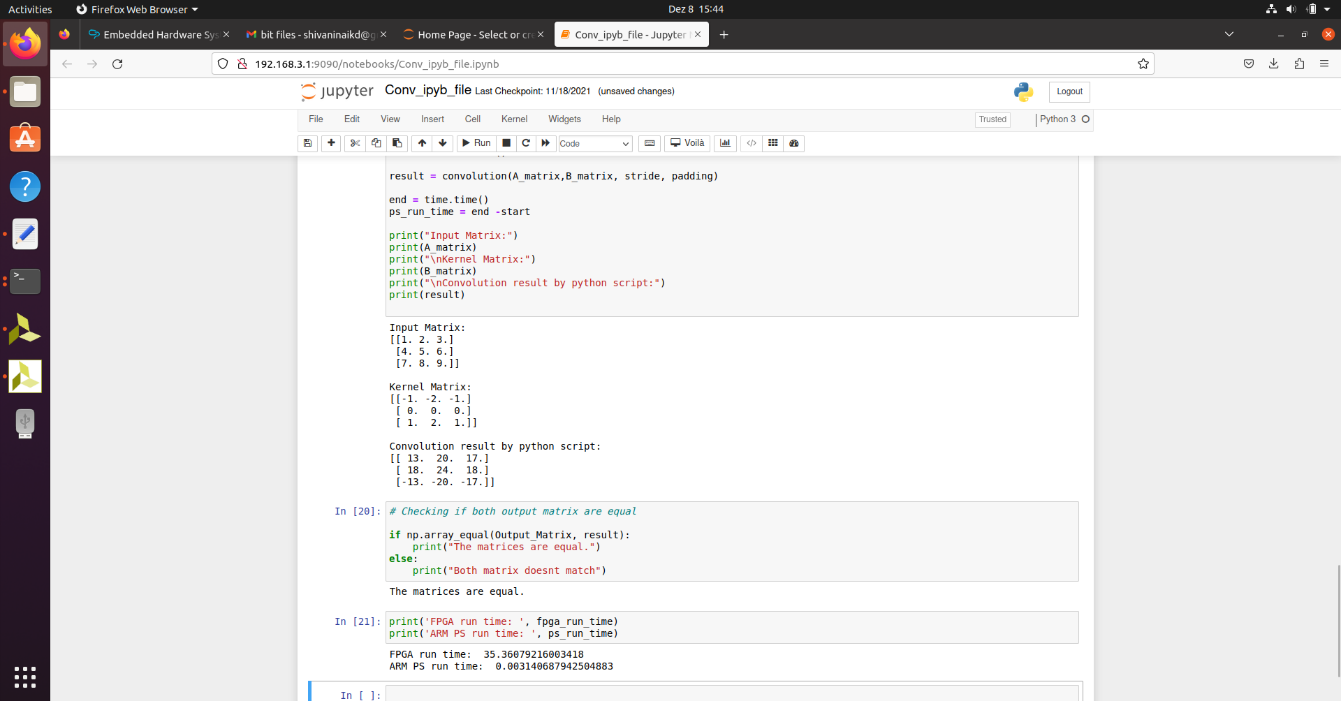
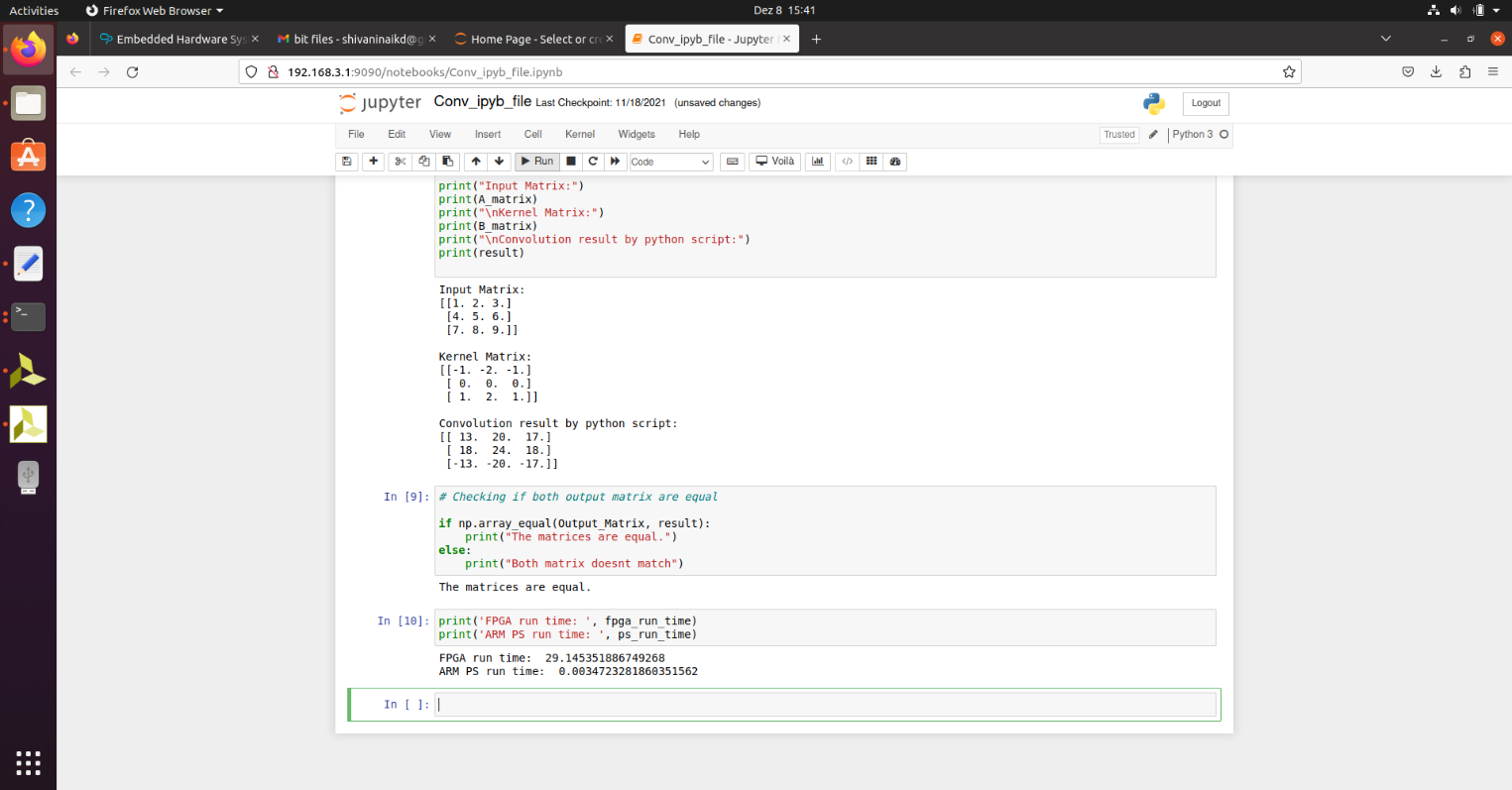


Figure-2 Timing and power report un-optimized and optimized design

* Uploading the .bit and .hwh files in Jupiter notebook and running the python script.
* The .py scripts first send the image and kernal data via send channel and receive output in output buffer. Then convolution function implemented in python using numpy library calculates the output.
* Both results are compared for equality. The end result are as follows.



Result of Unoptimized bit file

Result of Optimized bit file

**Observation and Conclusion**

* Pragmas were added to reduce the critical path and latency. It significantly **reduced the latency** by 81% as seen in table-1.
* We can also observe from table-1 that **no. of loop iteration also significantly decreased** by using Loop flatten and merge pragma.
* But then **resource utilization increased** in the optimised case that means overall design area increased too.
* Total on-chip **power increased slightly** as well. Which was expected as no of resource increased, they would utilize more power. Refer figure-2.
* So, there is always trade-off between timing, power and area. All cannot be optimised at the same time. In this project timing was prioritised over others.

**Submitted by,**

Kishan Dadhania

Matriculation no. - 5112720