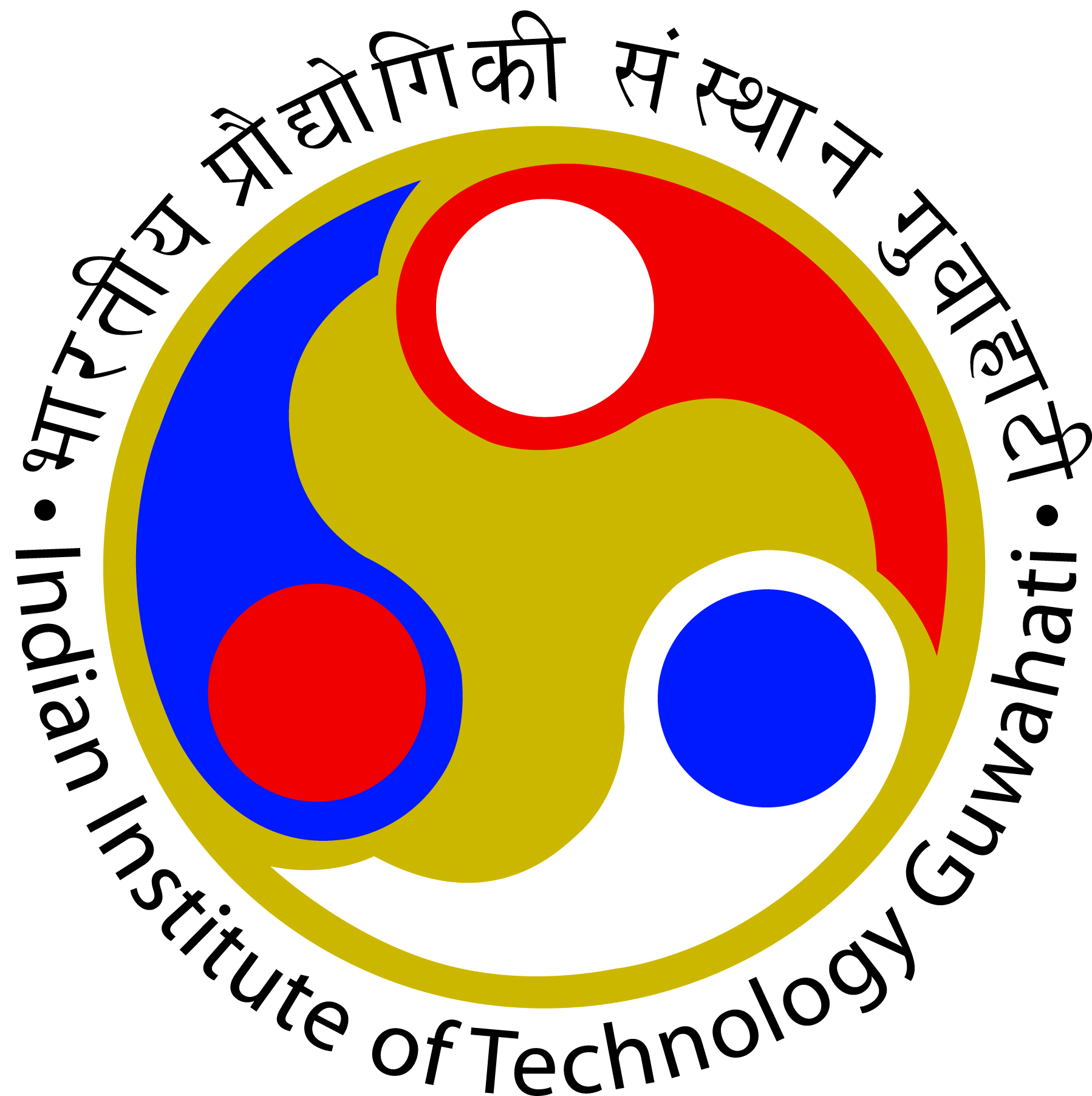
****

**HLS for Hand-Written Digit Recognition**

**Group Number: 26**

|  |  |
| --- | --- |
| **Group member** | **Roll Number** |
| Rupnarayan Kumar | 234101046 |
| Sunny Priyadarshi | 234101051 |
| Manish Karmakar | 234101028 |
| Mukesh Barpete | 234101030 |
| Kumar Sanu | 234101025 |

Guided By: **Dr. Chandan Karfa**

Allocated TA: **Manoj Choudhary**

***1.Description of the model : -***

**Task : -** This model trains a convolutional neural network (CNN) using the MNIST dataset for handwritten digit recognition.

**Dataset :-** MNIST training dataset, consisting of 60,000 grayscale images of handwritten digits.

**Layer Description : -**

* Input Layer: The model takes grayscale images of size 28x28 pixels as input.
* Convolutional Layers :- The convolutional layer has 16 filters of size 3x3, using the ReLU activation function. In the convolutional layer:

(3 \* 3 \* 1 + 1) \* 16 = 160 parameters.

* Pooling Layer : - A max-pooling layer with a pool size of 2x2 follows the convolutional layer. Here no parameters.
* Flatten Layer :- Flattens the output from the convolutional layers into a 1D vector. Here there are no parameters to learn.
* Dense Layer :- A fully connected layer with 10 units and the softmax activation function, which outputs probabilities for each digit class (0-9).

In the output dense layer: (13\*13\*16) \* 10 = 27040 parameters

* Output Layer : - generating the probability distribution over the possible digit classes and classify in (0-9)class.

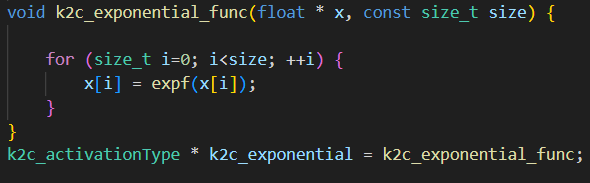
***2. Changes made to make keras2c generated files synthesizable and a brief description of the change made.***

**A. Removed Function’s pointer**

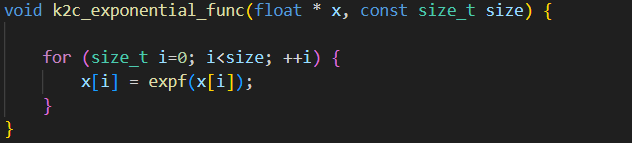
* In HLS, function pointers can pose challenges because HLS tools need to analyse and synthesise the code into hardware. Function pointers introduce dynamic behaviour, making it harder for the HLS tool to determine the hardware structure and behaviour accurately.

Solution: we have taken the functions inside the main file.

Before:



After:



**B. Removed Pointer to Pointer access**

* Pointer to Pointer Access is causing the error of dynamic size array which is not supported by vivado synthesis.

Solution: Flattened the structure shown as below:

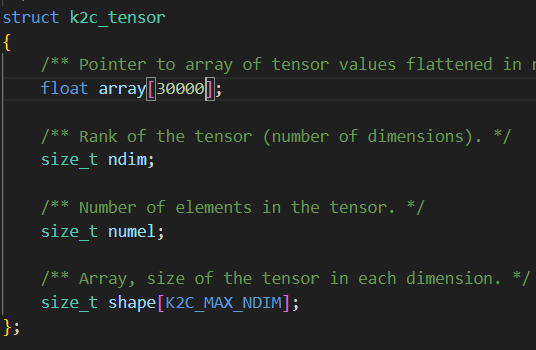
Before:



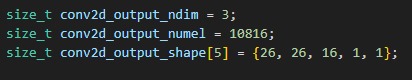
1st Optimization:

A screen shot of a computer code

Description automatically generated



2nd Optimization



**C. Removed Dynamic Functions(memcpy)**

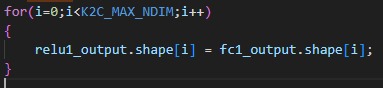
* memcpy() is not directly supported in HLS primarily because it's a library function that involves complex dynamic memory operations, which are challenging to synthesise into hardware efficiently. Here's why memcpy() isn't directly supported in HLS:

Solution: Rewritten in the form of a loop.

Before:



After:



**D. Removed Segmentation Fault error while Simulation**

Solution: Took all the arrays outside the functions.

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

Solution: We have not removed any pragmas while generating the HLS4ML report.

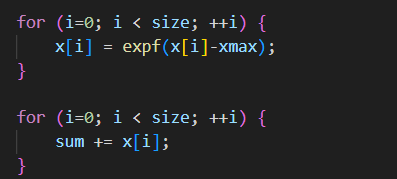
***4. Mention all the issues that are faced(dependencies and versions) and solutions to resolve.***

A black and white screen with white text

Description automatically generated

***5.Optimizations: For each optimization applied (pragma), justify why it has been used.***

Loop merging, reduced array access and pipeline.

******

***A computer screen with text and numbers

Description automatically generated***

***Reduced array access and applied pipeline***

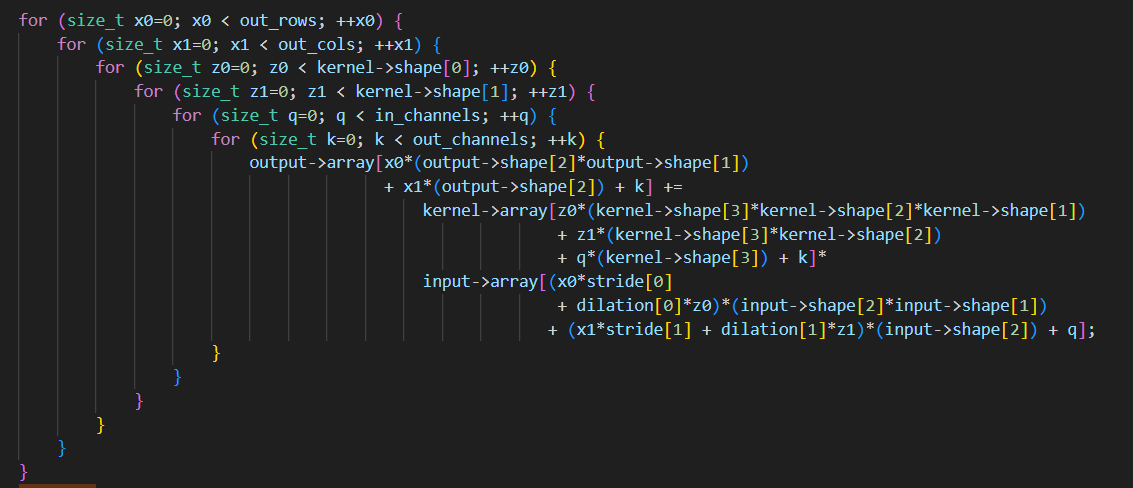
***A screen shot of a computer code

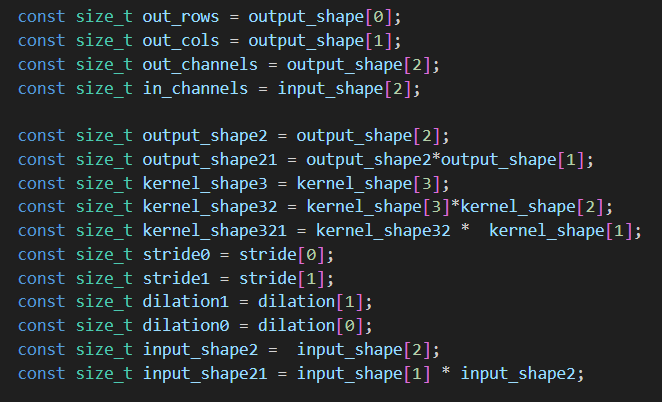
Description automatically generated***

***A black screen with white text

Description automatically generated***

***Code Motion***

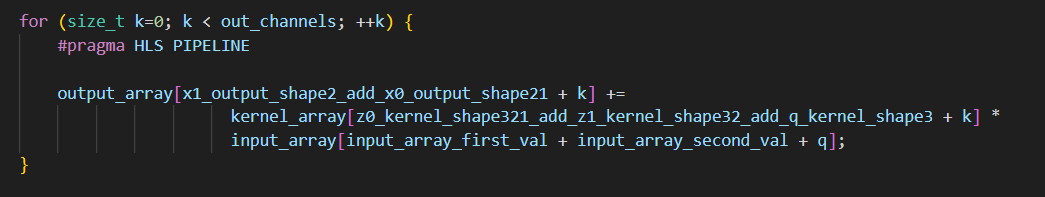
******

******

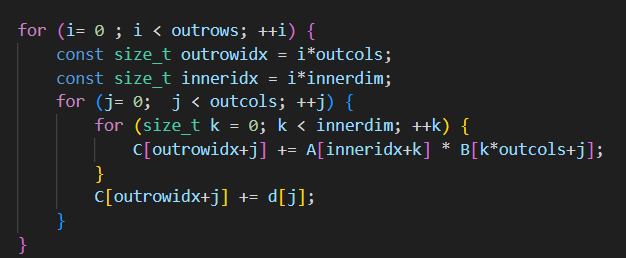
***A computer screen shot of text

Description automatically generated***

***And then applied pipeline***

******

***Loop tiling***

******

***A screen shot of a computer program

Description automatically generated***

***Inline small functions***

a) k2c\_relu\_function

b) k2c\_bias\_add

c) k2c\_add\_bias

d) k2c\_flatten

1. K2c\_softmax\_func

* Loop1: minimised array access by code motion.
* Loop2: minimised array access by code motion and applied pipeline. Also merged loop2 and loop3 as it runs same number of iterations.

Array access and exponential computation takes much clocks , which can be done parallely, so we applied pipeline.

* Loop4: minimised array access and applied pipeline.

Multiplier takes more cycle and assignment takes 1 cycle. So, we can apply pipelining and save clocks.

1. K2c\_relu\_func

* Loop1: array access, condition checking then assignment which takes 3 cycles. We can apply pipeline and save latency.

1. K2c\_bias\_add

* Loop1: array access, addition and assignment. By applying pipelining we can save latency.

1. K2c\_conv2d

* Loop1: used unroll factor=2. For array assignment
* Loop2: It has 6 nested loop and inside loops there are much array access and computation and those arrays are not updating. So, we can minimise array access and computation by applying code Motion.

In the innermost loop, we used pipeline to save cycles in computation.

1. K2c\_maxpool2d

* There are much array access and computation on array which are not updating so we can save array access and computation by applying code motion. In the inner loop array access, condition checking then assignment which can be parallelized somewhat. So, we applied pipelining.

1. K2c\_affine\_matmul

* Here matrix multiplication is being calculated. So we can apply loop tiling to increase locality of reference.

1. K2c\_idx2sub

* Here we minimised array access and three computations are being done which can be parallelised . So we used pipeline.

1. K2c\_sub2idx

* Here array access and two operations are being done. We can parallelise the computation so applied pipeline here

1. K2c\_matmul

* Array access and computations are being done. Computations are constant so we applied code motion, and some computations can be parallelised so applied pipelining.

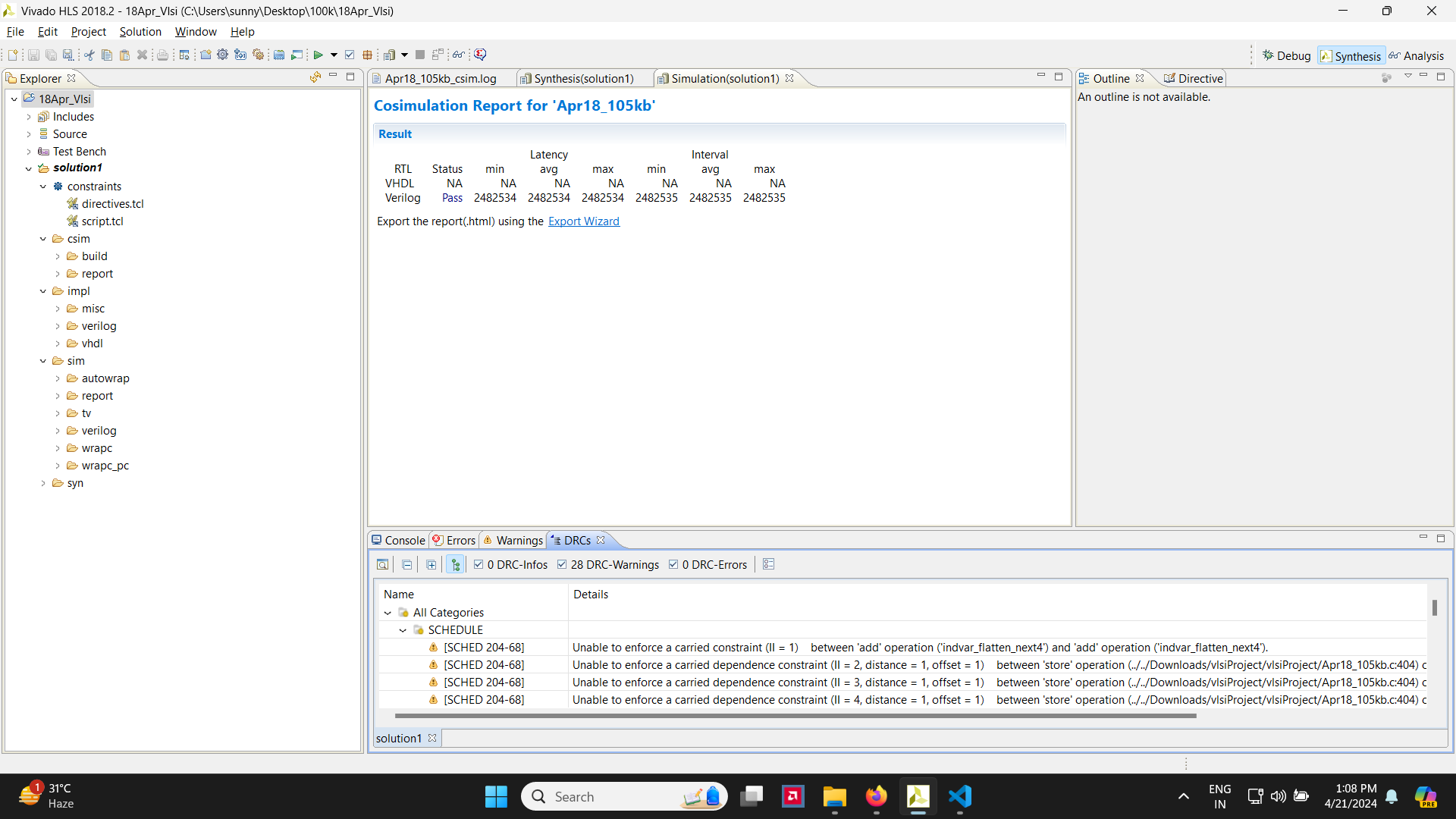
1. K2c\_dot

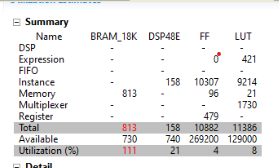
* Here multiple array access and computations were done. So, to parallelise the computation we used pipelining.

Here redundant array access and computations were being done, so we used code motion and pipelining.

**6. Results:**

* **Latency and area overhead table for Baseline (Unoptimized).**





* **Latency and area overhead table for Optimized**

A screenshot of a report

Description automatically generated

A screenshot of a computer

Description automatically generated

* **HLS4ML generated Latency and area overhead table.**

**A screen shot of a computer

Description automatically generated**

**A black screen with white text

Description automatically generated**

* **Comparison report of both Optimized and HLS4ML generated report.**

HLS4ML Table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Design | LUT | FF | DSP | BRAM | Latency(min/max) | Clock Period |
| xcvu13p-flga2577-2-e | 214967 | 83326 | 1 | 311 | 32924 / 32929 | 5 |

Vivado Optimized Table:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Design | LUT | FF | DSP | BRAM | Latency(min/max) | Clock Period/Estimated |
| Unoptimized | Artix7 | 11386 | 10882 | 158 | 813 | 24825534/24825534 | 10/8.345 |
| Final Optimized | Artix7 | 51467 | 56316 | 150 | 241 | 135764 / 135764 | 5/5.681 |

Thank You