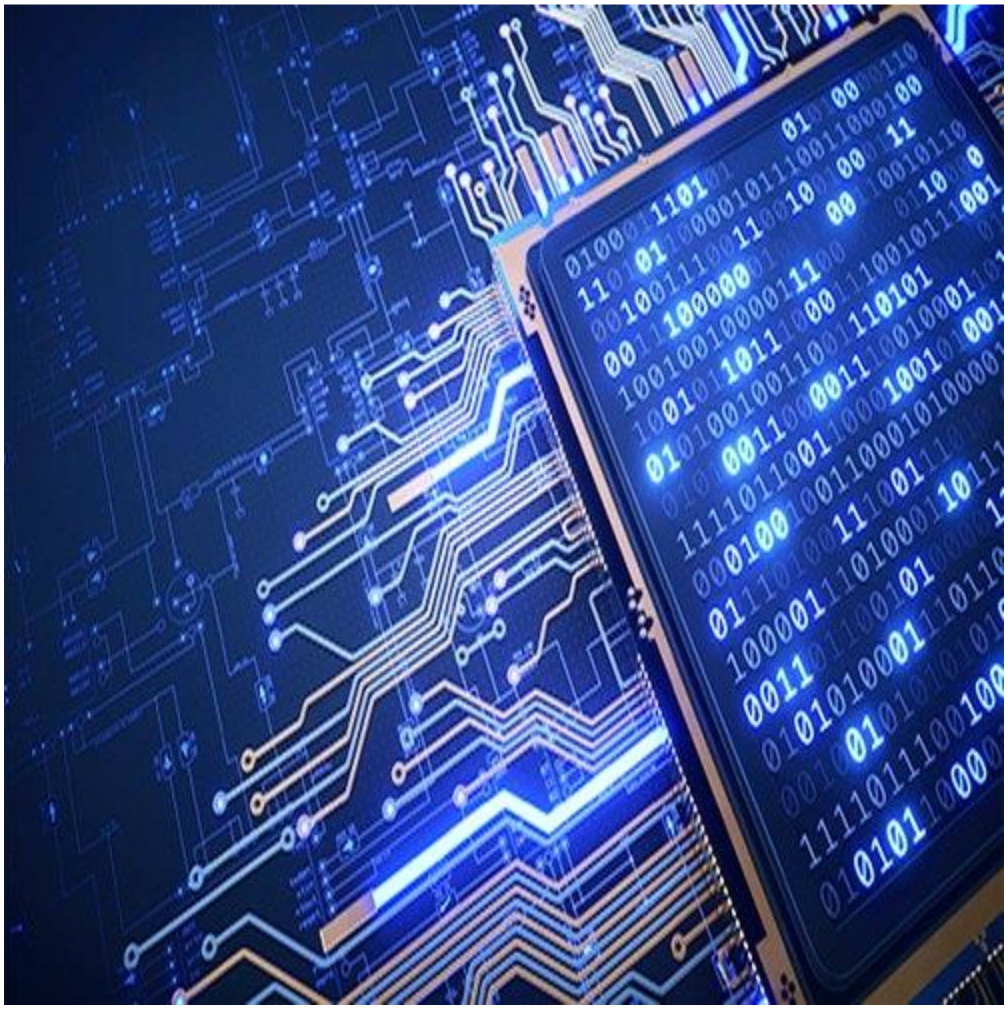


2d CONVOLUTION

EMBEDDED HARDWARE SYSTEMS DESIGN PRACTICAL
WS 2023/24

Sachhidanand | 5116774 | Nano-Electronic Systems

1. INTRODUCTION

Convolutional neural networks, or CNNs, have become extremely effective tools for a variety of computer vision applications in recent years, such as object identification, image segmentation, and image classification. There is a growing need to use hardware platforms to accelerate these computationally demanding CNNs as the demand for low-latency and real-time processing grows. The Multiprocessor System-on-Chip is one such platform that provides a balance between performance and flexibility (MPSoC).

The aim of this project is to utilise Xilinx Vivado to implement the process of convolution on an MPSoC. Multiple processing units offered by the MPSoC architecture can be used to optimise and parallelize CNN computations, significantly speeding up the process in comparison to typical software implementations.

The Convolution filter is designed using Xilinx's Vitis HLS IDE. High level language support is offered by Vitis HLS for both module design and verification. Additionally, it supports pragma directives, which tell the compiler how to convert kernel source code into RTL code. The RTL code can be ported using the export RTL function and then imported as an IP block into the Vivado IDE library to facilitate additional design work. An FPGA known as the **Xilinx Ultra 96v2** onboarding a chip **part no.: xczu3eg-sbva484-1-i** serves as the experiment's hardware. The Zynq ARM core is subsequently attached to the imported IP block to facilitate data transmission and reception. AXI-Stream has been selected as the interface for communicating with the ARM core. The final algorithm is tested using a Python Script in Jupyter Notebook.

2. 2D CONVOLUTION

A basic operation in convolutional neural networks (CNNs) and image processing is the 2D convolution operation. The basic architecture for convolution algorithm includes the following parameters –

1) **INPUT IMAGE and KERNEL** - Start with a 2D convolutional kernel (also called a filter or mask) and a 2D input matrix, which frequently represents an image.

2) **PADDING** - To make sure that the convolution operation covers the entire input, you can optionally pad the input matrix with zeros or other values. In order to preserve spatial dimensions, padding is particularly helpful near the input's edges.

3) **STRIDE -** Convolutional kernels are moved to the next position in the input matrix by a stride, which is a fixed step size.At every position, repeat the convolution operation to create the feature map, a 2D output matrix.

Convolution operation include following steps –

- The convolutional kernel is positioned above the input matrix. The kernel is always smaller than the input matrix.
- Perform an element-wise multiplication between the elements of the input matrix covered by the kernel and the corresponding elements of the kernel.

- Sum up the results of the element-wise multiplications to obtain a single scalar value. This scalar is the result of the convolution operation at a specific location (usually the top-left corner of the kernel's position).

The size of the output feature map is determined by the dimensions of the input matrix, the size of the kernel, the padding applied, and the stride used during the convolution operation.

$$\text{output width} = \frac{W - F_w + 2P}{S_w} + 1$$

$$\text{output height} = \frac{H - F_h + 2P}{S_h} + 1$$

Where W and H are input image width and height respectively, Fw and Fh is the width and height of the filter, P is the padding given and S is the stride.

3. PROPOSED ALGORITHM OF 2d CONVOLUTION FOR IMPLEMENTATION

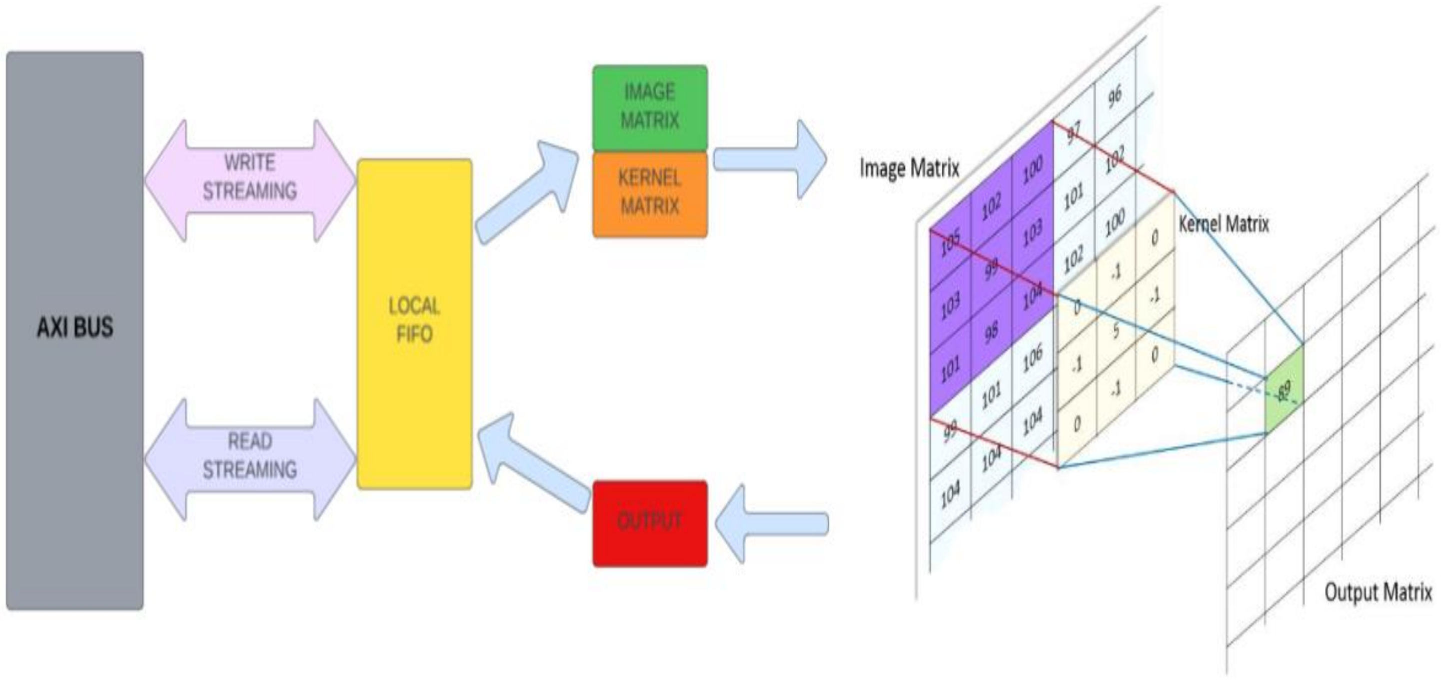
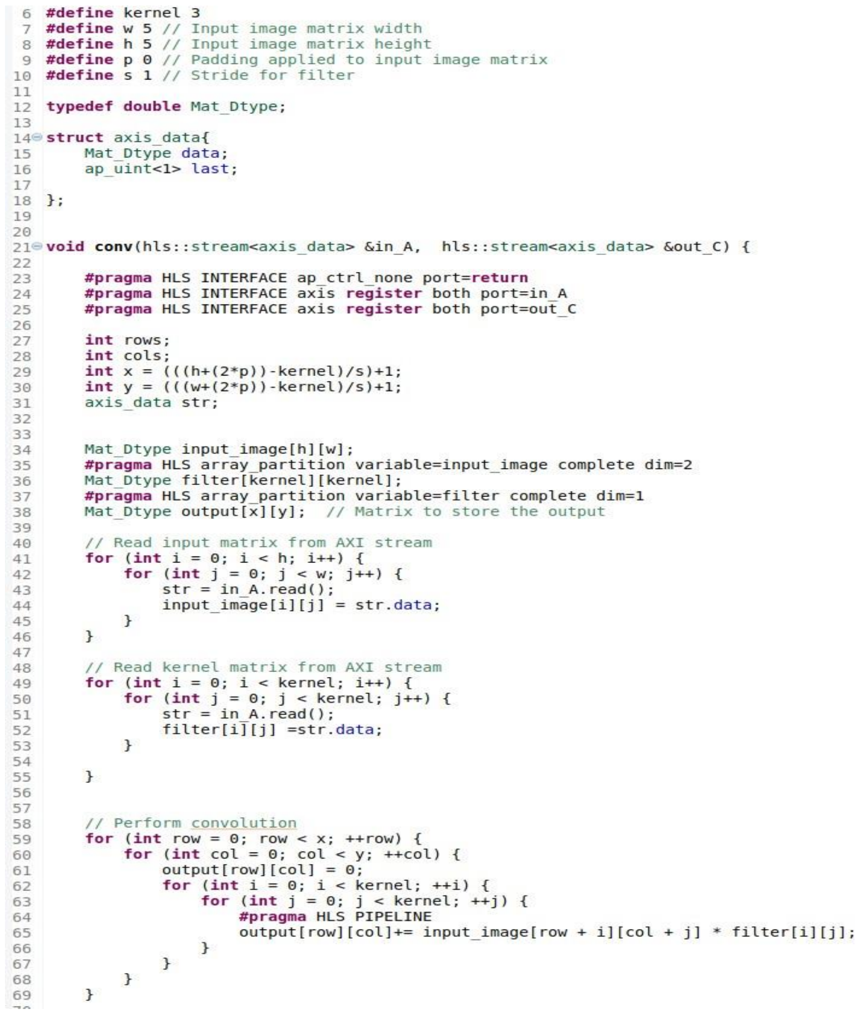


Figure 1 - Complete design block diagram

3.1.HLS code without optimization



```
#define kernel 3
#define w 5 // Input image matrix width
#define h 5 // Input image matrix height
#define p 0 // Padding applied to input image matrix
#define s 1 // Stride for filter

typedef double Mat_Dtype;

struct axis_data{
    Mat_Dtype data;
    ap_uint<1> last;

};


void conv(hls::stream<axis_data> &in_A,  hls::stream<axis_data> &out_C) {

    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE axis register both port=in_A
    #pragma HLS INTERFACE axis register both port=out_C

    int rows;
    int cols;
    int x = (((h+(2*p))-kernel)/s)+1;
    int y = (((w+(2*p))-kernel)/s)+1;
    axis_data str;


    Mat_Dtype input_image[h][w];
    #pragma HLS array_partition variable=input_image complete dim=2
    Mat_Dtype filter[kernel][kernel];
    #pragma HLS array_partition variable=filter complete dim=1
    Mat_Dtype output[x][y];  // Matrix to store the output

    // Read input matrix from AXI stream
    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            str = in_A.read();
            input_image[i][j] = str.data;
        }
    }

    // Read kernel matrix from AXI stream
    for (int i = 0; i < kernel; i++) {
        for (int j = 0; j < kernel; j++) {
            str = in_A.read();
            filter[i][j] =str.data;
        }

    }


    // Perform convolution
    for (int row = 0; row < x; ++row) {
        for (int col = 0; col < y; ++col) {
            output[row][col] = 0;
            for (int i = 0; i < kernel; ++i) {
                for (int j = 0; j < kernel; ++j) {
                    #pragma HLS PIPELINE
                    output[row][col]+= input_image[row + i][col + j] * filter[i][j];
                }
            }
        }
    }
```

Figure 2 - C++ code for 2d Convolution (without optimisation)

1. **Mat_Dtype data**: This member is a representation of the data that is actually being processed or sent. The precise type (Mat_Dtype) would be defined in another part of the code; it can be a user defined custom data type that holds values for pixels, matrix elements, or any other pertinent information.
2. **ap_uint<1> last**: In the AXI Stream interface, this member usually represents a control signal. The final signal in an AXI stream is frequently used to signal the conclusion of a packet or frame. The current data packet is the last in a series of packets or frames when last is asserted (set to 1). When streaming, this is essential for distinguishing between various data sets or frames.

NOTE: - Therefore, the **axis_data** struct appears to encapsulate both the actual data being transported or processed (Mat_Dtype data) and a control signal (ap_uint1> last) signaling the end of a packet/frame within the AXI Stream interface. This struct can be used to determine the format of data provided or received over the AXI Stream and helps organize the information for processing in your Vivado HLS-based 2D convolution algorithm.

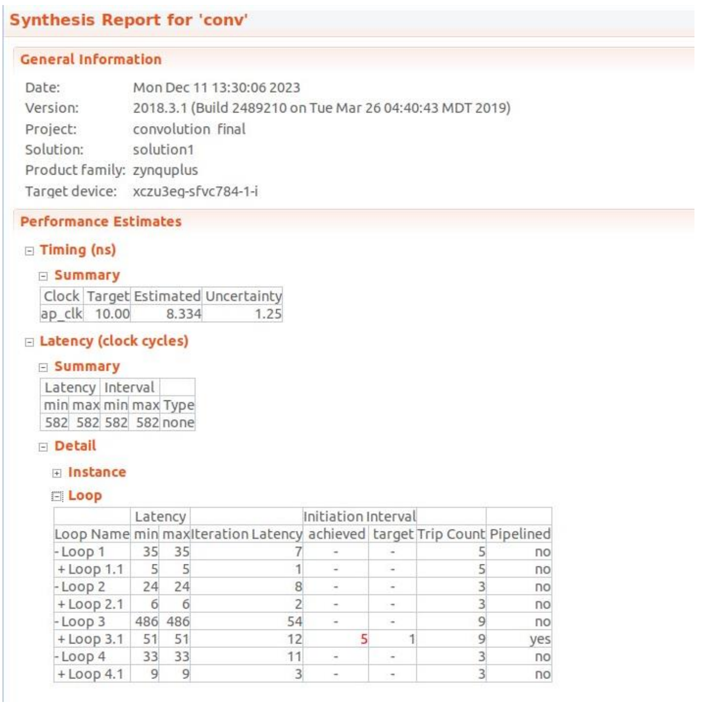
The array, **input_image**, is of type Mat_Dtype and has dimensions of 'w' and 'h'. This array is optimized for hardware synthesis using the **#pragma HLS array_partition** directive. The specification "complete dim=2" means that the array has to be divided in half along the second dimension (dim=2). This could allow for concurrent access to the array's columns in hardware since each element in the second dimension will be handled as a distinct hardware resource.

The second input array, **filter**, is of dimension ‘kernel’ and ‘kernel’ of type Mat_Dtype, is also a 2D array. Optimization for this array is specified by the **#pragma HLS array_partition** directive where “complete dim=1” denotes a full partition of the array along the first dimension (dim=1). Thus, every component in the first dimension will be treated as a separate hardware resource, potentially allowing parallel access to rows of the array in hardware.

Both pragmas (array_partition) divide the arrays into smaller partitions in an effort to efficiently utilize the hardware resources. Through increased parallelism in the hardware design developed, this optimization may help improve performance by facilitating more effective data processing.

Nonetheless, the particular hardware architecture, design limitations, and intended optimizations for the 2D convolution process you're implementing in Vivado HLS should all be taken into account when selecting complete partitioning and the dimension along which to partition (dim=1 or dim=2).

THE RESULTS AFTER HLS SYSNTHESIS FOR THE NON-OPTIMIZED CODE ARE SHOWN BELOW -



Synthesis Report for 'conv'

General Information

Date:	Mon Dec 11 13:30:06 2023
Version:	2018.3.1 (Build 2489210 on Tue Mar 26 04:40:43 MDT 2019)
Project:	convolution final
Solution:	solution1
Product family:	zynquplus
Target device:	xczu3eg-sfvc784-1-i

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.334	1.25

Latency (clock cycles)

Summary

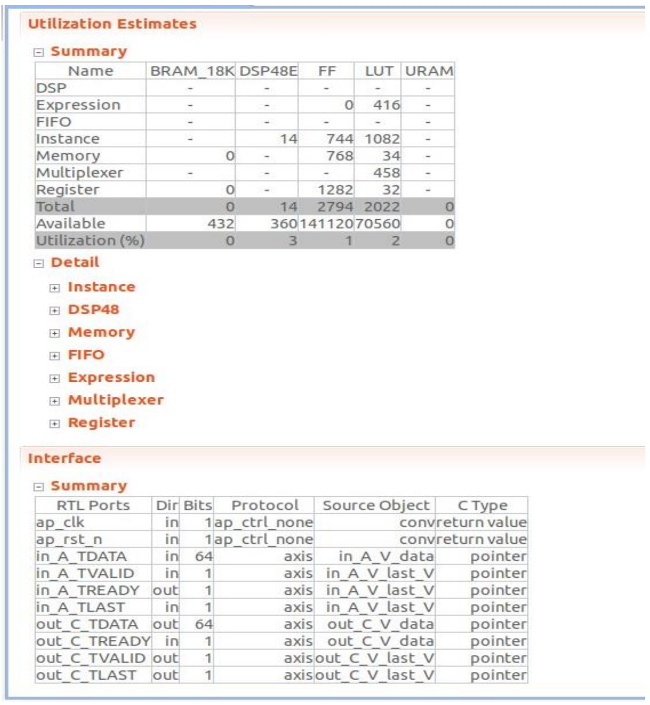
Latency		Interval		
min	max	min	max	Type
582	582	582	582	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	35	35	7	-	-	5	no
+ Loop 1.1	5	5	1	-	-	5	no
- Loop 2	24	24	8	-	-	3	no
+ Loop 2.1	6	6	2	-	-	3	no
- Loop 3	486	486	54	-	-	9	no
+ Loop 3.1	51	51	12	5	1	9	yes
- Loop 4	33	33	11	-	-	3	no
+ Loop 4.1	9	9	3	-	-	3	no



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	416	-
FIFO	-	-	-	-	-
Instance	-	14	744	1082	-
Memory	0	-	768	34	-
Multiplexer	-	-	-	458	-
Register	0	-	1282	32	-
Total	0	14	2794	2022	0
Available	432	360	141120	70560	0
Utilization (%)	0	3	1	2	0

Detail

Instance

DSP48

Memory

FIFO

Expression

Multiplexer

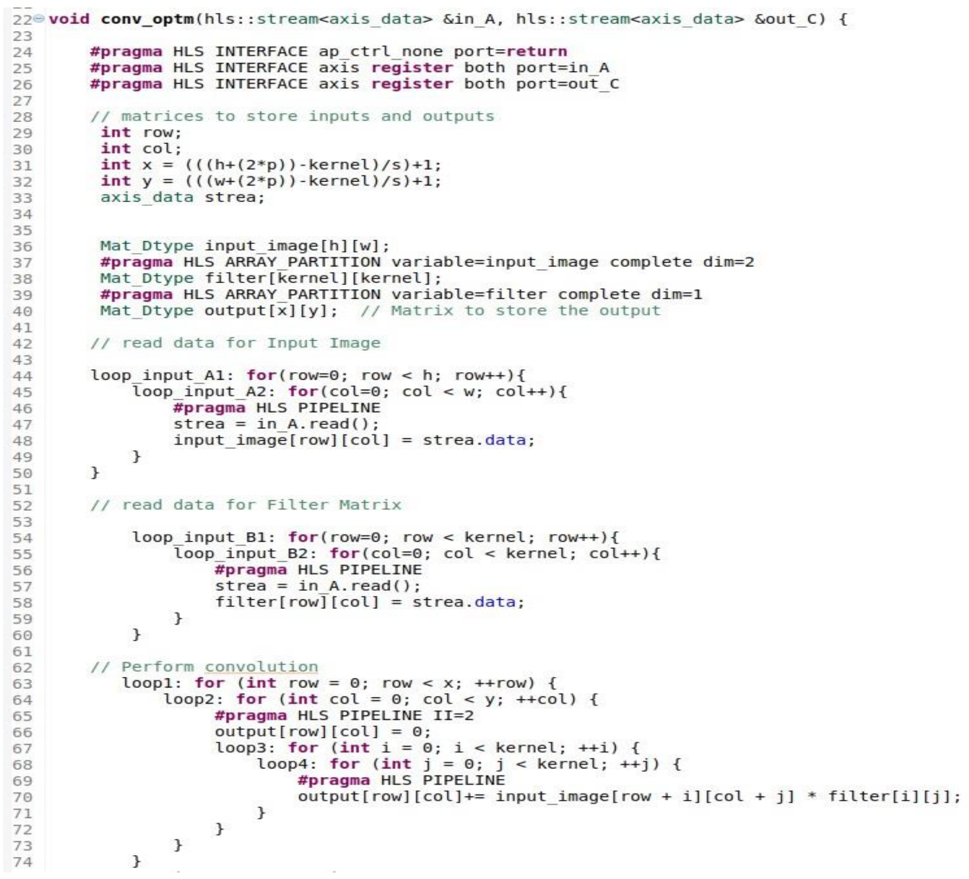
Register

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	conv	return value
ap_rst_n	in	1	ap_ctrl_none	conv	return value
in_A_TDATA	in	64	axis	in_A_V_data	pointer
in_A_TVALID	in	1	axis	in_A_V_last_V	pointer
in_A_TREADY	out	1	axis	in_A_V_last_V	pointer
in_A_TLAST	in	1	axis	in_A_V_last_V	pointer
out_C_TDATA	out	64	axis	out_C_V_data	pointer
out_C_TREADY	in	1	axis	out_C_V_data	pointer
out_C_TVALID	out	1	axis	out_C_V_last_V	pointer
out_C_TLAST	out	1	axis	out_C_V_last_V	pointer

3.2. HLS code with optimization



```
void conv_optm(hls::stream<axis_data> &in_A, hls::stream<axis_data> &out_C) {

    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE axis register both port=in_A
    #pragma HLS INTERFACE axis register both port=out_C

    // matrices to store inputs and outputs
     int row;
     int col;
     int x = (((h+(2*p))-kernel)/s)+1;
     int y = (((w+(2*p))-kernel)/s)+1;
     axis_data strea;


     Mat_Dtype input_image[h][w];
     #pragma HLS ARRAY_PARTITION variable=input_image complete dim=2
     Mat_Dtype filter[kernel][kernel];
     #pragma HLS ARRAY_PARTITION variable=filter complete dim=1
     Mat_Dtype output[x][y];  // Matrix to store the output

    // read data for Input Image

    loop_input_A1: for(row=0; row < h; row++){
        loop_input_A2: for(col=0; col < w; col++){
            #pragma HLS PIPELINE
            strea = in_A.read();
            input_image[row][col] = strea.data;
        }
    }

    // read data for Filter Matrix

        loop_input_B1: for(row=0; row < kernel; row++){
            loop_input_B2: for(col=0; col < kernel; col++){
                #pragma HLS PIPELINE
                strea = in_A.read();
                filter[row][col] = strea.data;
            }
        }

    // Perform convolution
       loop1: for (int row = 0; row < x; ++row) {
           loop2: for (int col = 0; col < y; ++col) {
                #pragma HLS PIPELINE II=2
                output[row][col] = 0;
                loop3: for (int i = 0; i < kernel; ++i) {
                    loop4: for (int j = 0; j < kernel; ++j) {
                        #pragma HLS PIPELINE
                        output[row][col]+= input_image[row + i][col + j] * filter[i][j];
                    }
                }
            }
        }
```

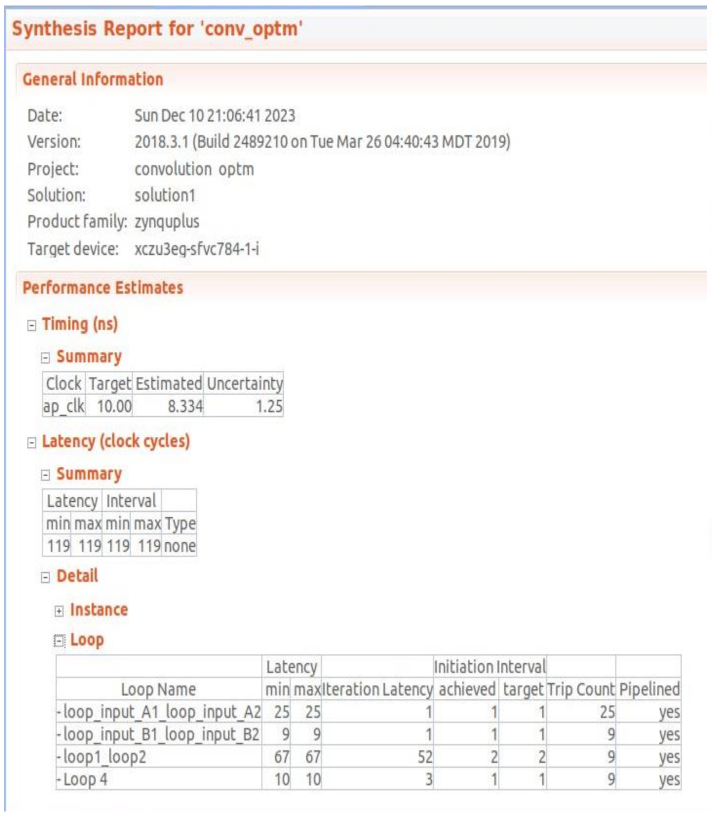
Figure 3 - Optimized C++ code for 2d Convolution

To instruct the tool to attempt to establish a pipeline in the created hardware for a loop or series of operations, use the **#pragma HLS pipeline** directive in Vivado HLS. By dividing an operation into phases that can run concurrently, a technique known as pipelining can increase a design's throughput.

The **#pragma HLS pipeline** directive instructs the tool to look for parallelism and overlapping operations when applied to a loop or a block of code in order to improve performance by increasing throughput and lowering the **Initiation Interval (II)**.

THE RESULTS AFTER HLS SYSNTHESIS FOR THE OPTIMIZED CODE ARE SHOWN BELOW –

- From the summary report we can see that there was an **4.89x increase in the latency after optimization**.



Synthesis Report for 'conv_optm'

General Information

Date:	Sun Dec 10 21:06:41 2023
Version:	2018.3.1 (Build 2489210 on Tue Mar 26 04:40:43 MDT 2019)
Project:	convolution_optm
Solution:	solution1
Product family:	zynquplus
Target device:	xczu3eg-sfvc784-1-i

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.334	1.25

Latency (clock cycles)

Summary

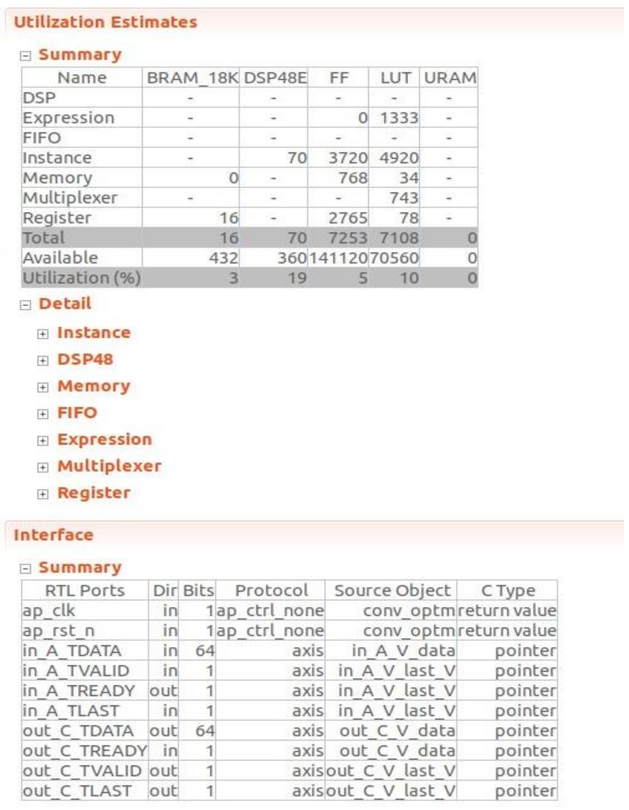
Latency		Interval		
min	max	min	max	Type
119	119	119	119	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop_input_A1_loop_input_A2	25	25	1	1	1	25	yes
- loop_input_B1_loop_input_B2	9	9	1	1	1	9	yes
- loop1_loop2	67	67	52	2	2	9	yes
- Loop 4	10	10	3	1	1	9	yes



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1333	-
FIFO	-	-	-	-	-
Instance	-	70	3720	4920	-
Memory	0	-	768	34	-
Multiplexer	-	-	-	743	-
Register	16	-	2765	78	-
Total	16	70	7253	7108	0
Available	432	360	141120	70560	0
Utilization (%)	3	19	5	10	0

Detail

Instance

DSP48

Memory

FIFO

Expression

Multiplexer

Register

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	conv_optm	return value
ap_rst_n	in	1	ap_ctrl_none	conv_optm	return value
in_A_TDATA	in	64	axis	in_A_V_data	pointer
in_A_TVALID	in	1	axis	in_A_V_last_V	pointer
in_A_TREADY	out	1	axis	in_A_V_last_V	pointer
in_A_TLAST	in	1	axis	in_A_V_last_V	pointer
out_C_TDATA	out	64	axis	out_C_V_data	pointer
out_C_TREADY	in	1	axis	out_C_V_data	pointer
out_C_TVALID	out	1	axis	out_C_V_last_V	pointer
out_C_TLAST	out	1	axis	out_C_V_last_V	pointer

4. **BLOCK DIAGRAM**

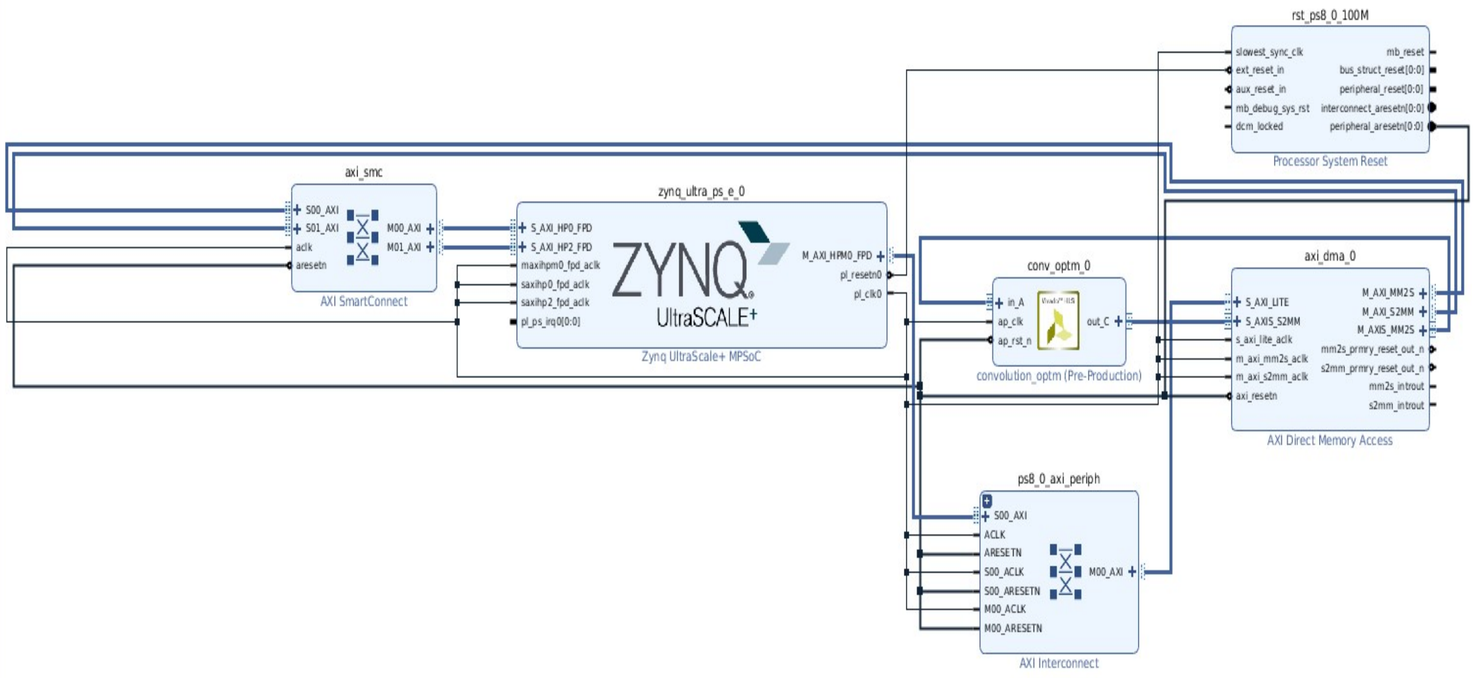


Figure 2 - Interface View

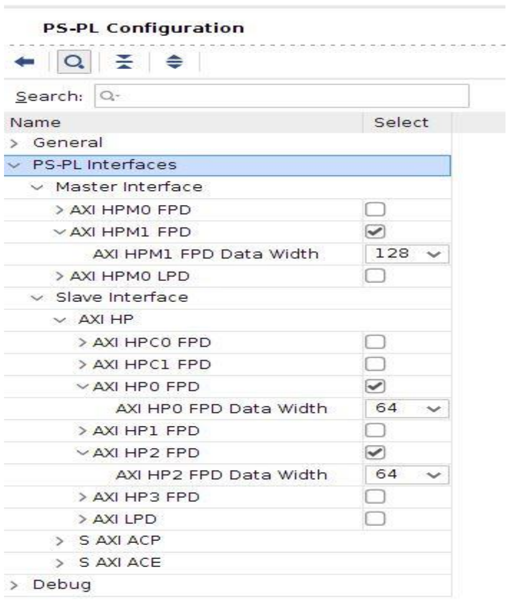
4.1 Zynq UltraScale+ MPSoC -

This intricate system combines processing (ARM Cortex-A cores) and programmable logic (FPGA fabric) on a single chip. It is appropriate for a variety of applications because it combines processing power with hardware programmability.

4.2 AXI Direct Memory Access –

A controller called AXI DMA (Direct Memory Access) is utilized to provide fast data transmission between the system's peripherals and memory. Large volumes of data can be moved between system components quickly and efficiently without using the CPU when this technique is applied.

Configuration of the Zynq UltraScale+ is as follows –

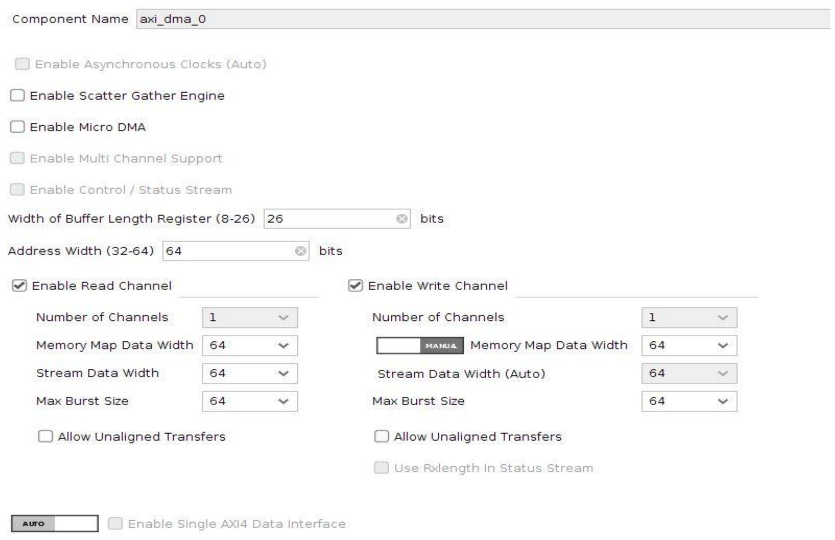


PS-PL Configuration

Search:

Name	Select
> General	
⌄ PS-PL Interfaces	
⌄ Master Interface	
> AXI HPM0 FPD	☐
⌄ AXI HPM1 FPD	✓
AXI HPM1 FPD Data Width	128
> AXI HPM0 LPD	☐
⌄ Slave Interface	
⌄ AXI HP	
> AXI HPC0 FPD	☐
> AXI HPC1 FPD	☐
⌄ AXI HP0 FPD	✓
AXI HP0 FPD Data Width	64
> AXI HP1 FPD	☐
⌄ AXI HP2 FPD	✓
AXI HP2 FPD Data Width	64
> AXI HP3 FPD	☐
> AXI LPD	☐
> S AXI ACP	
> S AXI ACE	
> Debug	

Configuration of the Zynq UltraScale+ is as follows –



Component Name: axi_dma_0

☐ Enable Asynchronous Clocks (Auto)

☐ Enable Scatter Gather Engine

☐ Enable Micro DMA

☐ Enable Multi Channel Support

☐ Enable Control / Status Stream

Width of Buffer Length Register (8-26): 26 bits

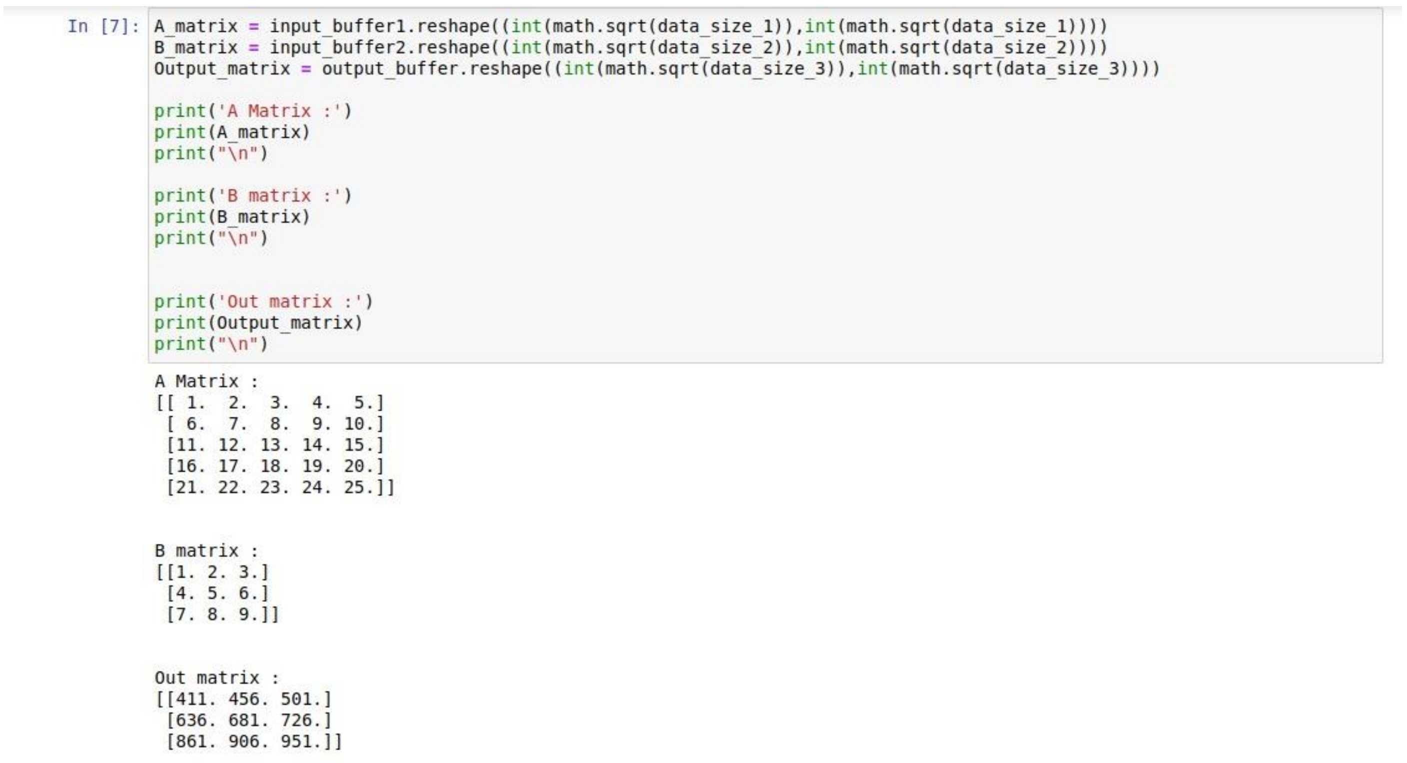
Address Width (32-64): 64 bits

✓ Enable Read Channel		✓ Enable Write Channel	
Number of Channels	1	Number of Channels	1
Memory Map Data Width	64	MANUAL Memory Map Data Width	64
Stream Data Width	64	Stream Data Width (Auto)	64
Max Burst Size	64	Max Burst Size	64
☐ Allow Unaligned Transfers		☐ Allow Unaligned Transfers	
		☐ Use Rxlength In Status Stream	

AUTO ☐ Enable Single AXI4 Data Interface

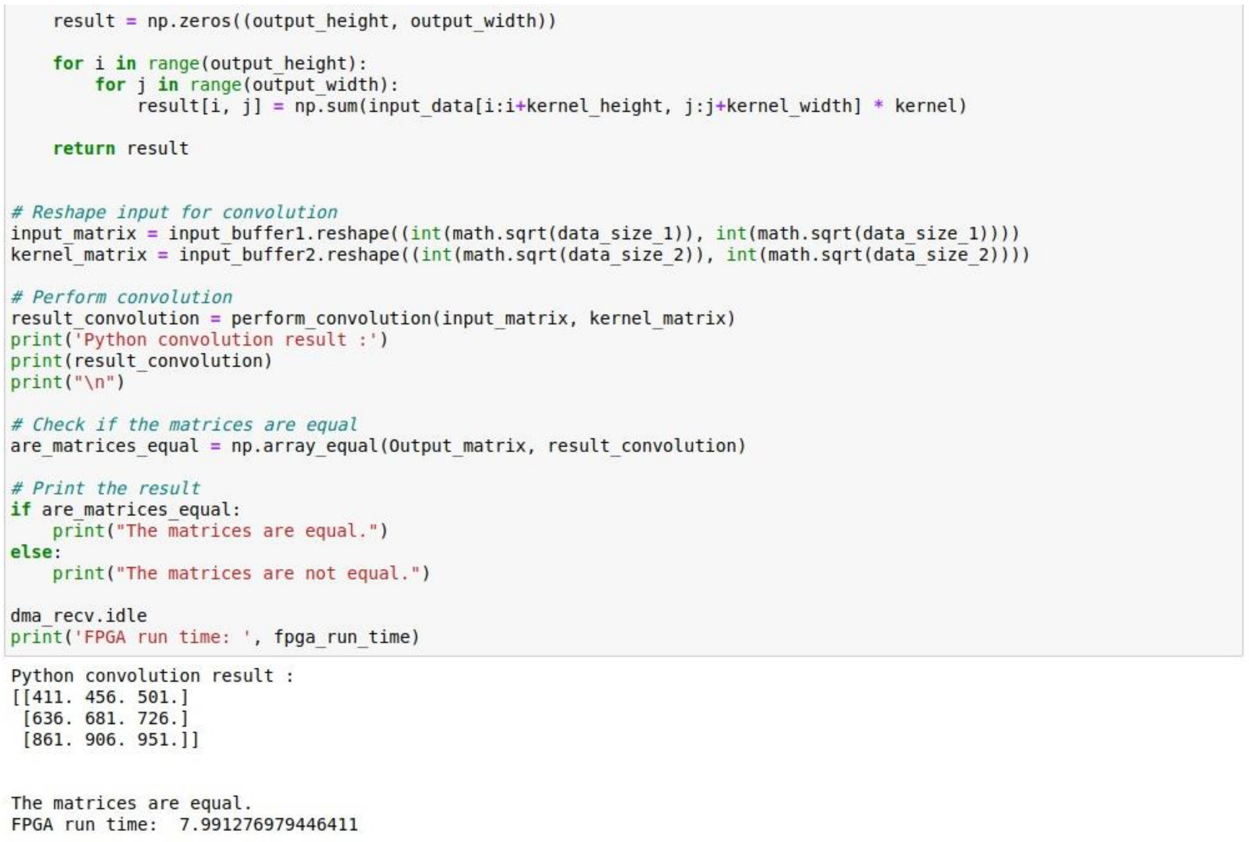
5. RESULTS

The final result of the above code after implementing on the MPSoC are –



```
In [7]: A_matrix = input_buffer1.reshape((int(math.sqrt(data_size_1)),int(math.sqrt(data_size_1))))
        B_matrix = input_buffer2.reshape((int(math.sqrt(data_size_2)),int(math.sqrt(data_size_2))))
        Output_matrix = output_buffer.reshape((int(math.sqrt(data_size_3)),int(math.sqrt(data_size_3))))

        print('A Matrix :')
        print(A_matrix)
        print("\n")

        print('B matrix :')
        print(B_matrix)
        print("\n")


        print('Out matrix :')
        print(Output_matrix)
        print("\n")
```

```
A Matrix :
[[ 1.  2.  3.  4.  5.]
 [ 6.  7.  8.  9. 10.]
 [11. 12. 13. 14. 15.]
 [16. 17. 18. 19. 20.]
 [21. 22. 23. 24. 25.]]


B matrix :
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]


Out matrix :
[[411. 456. 501.]
 [636. 681. 726.]
 [861. 906. 951.]]
```



```
    result = np.zeros((output_height, output_width))

    for i in range(output_height):
        for j in range(output_width):
            result[i, j] = np.sum(input_data[i:i+kernel_height, j:j+kernel_width] * kernel)

    return result


# Reshape input for convolution
input_matrix = input_buffer1.reshape((int(math.sqrt(data_size_1)), int(math.sqrt(data_size_1))))
kernel_matrix = input_buffer2.reshape((int(math.sqrt(data_size_2)), int(math.sqrt(data_size_2))))

# Perform convolution
result_convolution = perform_convolution(input_matrix, kernel_matrix)
print('Python convolution result :')
print(result_convolution)
print("\n")

# Check if the matrices are equal
are_matrices_equal = np.array_equal(Output_matrix, result_convolution)

# Print the result
if are_matrices_equal:
    print("The matrices are equal.")
else:
    print("The matrices are not equal.")

dma_recv.idle
print('FPGA run time: ', fpga_run_time)
```

```
Python convolution result :
[[411. 456. 501.]
 [636. 681. 726.]
 [861. 906. 951.]]


The matrices are equal.
FPGA run time:  7.991276979446411
```