ID1000500A
CONVOLUTIONER IP-CORE USER MANUAL

1. DESCRIPTION

The Convolutioner IP-core is a specialized processing system designed for convolution operations on multi-dimensional data, commonly used in signal processing and neural network applications.

Upon receiving a start command, this IP-core performs convolution by sliding a kernel over the input data and producing the corresponding output. The kernel size and stride can be configured via software, allowing for flexibility in various application requirements.

1.1. CONFIGURABLE FEATURES

Software configurations	Description
Convolution data	Data from extern memories (X, Y) to be process.

1.2. TYPICAL APPLICATION

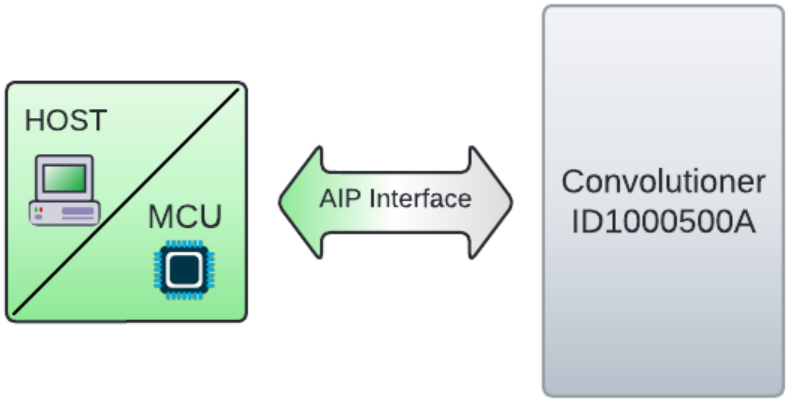

Figure 1.1 IP Convolutioner connected to a host

The Convolutioner IP-core is typically used in the following applications:

Image Processing

In image processing, the convolution is utilized for tasks such as edge detection, blurring, and sharpening. Edge detection involves identifying significant intensity changes within images to highlight boundaries, using filters like Sobel, Prewitt, and Canny. Blurring and smoothing reduce noise and details in images through kernels like Gaussian blur, while sharpening enhances image details by emphasizing high-

frequency components with kernels such as the Laplacian filter. These operations are fundamental in improving image quality and extracting meaningful information from visual data.

Signal Processing

In signal processing, the convolution is employed for filtering and feature extraction. Filtering involves removing unwanted components or noise from signals, achieved by designing various filters such as low-pass, high-pass, band-pass, and band-stop. Feature extraction is essential for applications like speech recognition and biomedical signal analysis (e.g., ECG, EEG), where convolution helps in identifying important characteristics within signals. These processes enhance signal clarity and enable the extraction of relevant data for further analysis.

Deep Learning

The convolution is crucial in deep learning, particularly in convolutional neural networks (CNNs). CNNs are widely used in computer vision tasks, where convolution layers extract spatial hierarchies of features from input images. This capability supports image classification, object detection, image segmentation, and feature map extraction. By facilitating these complex operations, the convolution enhances the performance of deep learning models in recognizing and understanding visual patterns and objects.

Medical Imaging

In medical imaging, the convolution plays a significant role in enhancing and analyzing images from modalities such as CT scans, MRI, and ultrasound. convolution operations help in tasks like tumor detection, organ segmentation, and image enhancement, providing clearer and more detailed images for diagnostic purposes. This capability improves the accuracy and efficiency of medical diagnoses, aiding healthcare professionals in making informed decisions.

Autonomous Vehicles

Autonomous vehicles rely on the convolution for object recognition and detection, as well as scene understanding. By processing sensor data from cameras, the IP-core identifies and locates objects like pedestrians, vehicles, and traffic signs. Additionally, it helps build a comprehensive understanding of the vehicle's surroundings, facilitating safe and efficient navigation decisions. This real-time processing capability is vital for the development and operation of self-driving cars.

Augmented Reality (AR) and Virtual Reality (VR)

For augmented reality (AR) and virtual reality (VR) applications, the convolution enables real-time image processing and feature detection. Real-time image processing enhances the real-world view by overlaying digital information, while feature detection ensures accurate overlay and interaction of virtual objects with the physical world. This capability is crucial for creating immersive and interactive AR and VR experiences, improving user engagement and interaction.

Remote Sensing

In remote sensing, the convolution processes satellite, and aerial images for applications such as climate monitoring, agricultural assessment, and urban planning. Convolution operations help in analyzing these images to classify terrain types and land usage, providing valuable insights into environmental and urban

changes. This information supports decision-making in various fields, including environmental conservation and urban development.

Video Processing

The convolution is essential in video processing for motion detection and tracking, as well as video enhancement. Motion detection and tracking involve identifying and following moving objects within video streams, while video enhancement improves video quality by reducing noise, sharpening images, and stabilizing footage. These capabilities are important for surveillance, entertainment, and communication applications, ensuring high-quality and reliable video content.

2. CONTENTS

1. DESCRIPTION 1
1.1. CONFIGURABLE FEATURES 1
1.2. TYPICAL APPLICATION 1
Image Processing 1
Signal Processing 2
Deep Learning 2
Medical Imaging 2
Autonomous Vehicles 2
Augmented Reality (AR) and Virtual Reality (VR) 2
Remote Sensing 2
Video Processing 3
2. CONTENTS 4
2.1. List of figures 5
2.2. List of tables 5
3. INPUT/OUTPUT SIGNAL DESCRIPTION 6
4. THEORY OF OPERATION 7
5. AIP interface registers and memories description 8
5.1. Status register 8
5.2. Configuration delay register 9
5.3. Input data memory (Mem X) 9
5.4. Output data memory (Mem Y) 9
5.5. Output data memory 9
6. PYTHON DRIVER 10
6.1. Usage example 10
6.2. Methods 11
6.2.1. Constructor 11
6.2.2. writeDataX 11
6.2.3. writeDataY 11
6.2.4. readData 11

6.2.5. startIP 12
6.2.6. configurationRegister 12
6.2.7. enableINT 12
6.2.8. disableINT 12
6.2.9. status 12
6.2.10. waitINT 13
6.2.11. conv 13
7. C DRIVER 13
7.1. Usage example 13
7.2. Driver functions 14
7.2.1. id1000500A_init 14
7.2.2. id1000500A_writeData 14
7.2.3. id1000500A_readData 15
7.2.4. id1000500A_startIP 15
7.2.5. id1000500A_configurationRegister 15
7.2.6. id1000500A_enableINT 15
7.2.7. id1000500A_disableINT 16
7.2.8. id1000500A_status 16
7.2.9. id1000500A_waitINT 16
7.2.10. id1000500A_finish 16
7.2.11. id1000500A_clearStatus 16
7.2.12. id1000500A_conv 17

2.1. List of figures

Figure 1.1 IP Convolutioner connected to a host 1
Figure 5.1 IP Convolutioner status register 8
Figure 5.2 Configuration delay register. 9

2.2. List of tables

Table 1 IP Convolution input/output signal description 6

3. INPUT/OUTPUT SIGNAL DESCRIPTION

Table 1 IP Convolution input/output signal description

Signal	Bit width	Direction	Description
General signals			
clk	1	Input	System clock
rst_a	1	Input	Asynchronous system reset, low active
en_s	1	Input	Enables the IP Core functionality
AIP Interface			
data_in	32	Input	Input data for configuration and processing
data_out	32	Output	Output data for processing results and status
conf_dbus	5	Input	Selects the bus configuration to determine the information flow from/to the IP Core
write	1	Input	Write indication, data from the data_in bus will be written into the AIP Interface according to the conf_dbus value
read	1	Input	Read indication, data from the AIP Interface will be read according to the conf_dbus value. The data_out bus shows the new data read.
start	1	Input	Initializes the IP Core process
int_req	1	Output	Interruption request. It notifies certain events according to the configurated interruption bits.
Core signals			
data_X	8	Input	Input data from extern memory to be process
memX_addr	5	Output	Output data for send memory address to get it data
dataY	8	Input	Input data from extern memory to be process
memY_addr	5	Output	Output data for send memory address to get it data
dataZ	16	Output	Output data for processing convolution result
memZ_addr	6	Output	Output data for send memory address
writeZ	1	Output	Write instruction, data from the convolution process will be written in dataZ
config_in	32	Input	Input data for send as both data from data_X and data_Y as a single word

busy_out	1	Output	Output data for status check. If the process is still active it means busy_out is active
done_out	1	Output	Output data for status check. If the process is over, it means busy_out is active

4. THEORY OF OPERATION

The Convolutioner IP-core is a specialized hardware module designed to perform convolution operations on multi-dimensional data. This IP-core provides a highly efficient and configurable solution for executing these operations in hardware, offering significant performance advantages over software-based implementations.

Convolution is a mathematical operation that combines two sets of information. In the context of the Convolutioner IP-core, the operation typically involves a multi-dimensional input data array (such as an image or feature map) and a kernel (or filter). The kernel is a smaller array that is applied to the input data to produce a transformed output.

The convolution operation can be described by the following steps:

1. **Initialization**: The input data and kernel are loaded into their respective memory locations within the IP-core.
2. **Kernel Sliding**: The kernel slides over the input data array in a systematic way, typically from the top-left to the bottom-right. At each position, an element-wise multiplication is performed between the kernel and the corresponding subarray of the input data.
3. **Summation**: The results of these multiplications are summed to produce a single output value for each position of the kernel.
4. **Stride and Padding**: The stride defines the step size by which the kernel moves across the input data. Padding can be applied to the input data borders to control the spatial dimensions of the output.
5. **Output Generation**: The computed values are stored in the output memory, forming the output data array.

Mathematically, the convolution operation for a 2D input and kernel can be expressed as:

$$(\boldsymbol{I} * \boldsymbol{K})(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} I(x+i, y+j) \cdot K(i,j)$$

where $\boldsymbol{I}$ is the input data, $\boldsymbol{K}$ is the kernel, and $\boldsymbol{m}$ and $\boldsymbol{n}$ are the dimensions of the kernel.

Data Flow

1. **Input Data Load**: The input data is loaded into the input memory of the IP-core.
2. **Kernel Load**: The convolution kernel is loaded into the kernel memory.
3. **Convolution Execution**: Upon receiving a start command, the IP-core begins the convolution operation by sliding the kernel over the input data, performing multiplications, summing the results, and storing the output.

4. **Output Data Storage**: The final convolved data is stored in the output memory and can be retrieved by the system for further processing or analysis.

5. AIP interface registers and memories description

5.1. Status register

Config: STATUS
Size: 32 bits
Mode: Read/Write.

This register is divided in three sections, see

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
								Mask bits								**Status Bits**								**Interrupt/Clear Flags**							
Reserved								Reserved							MSK	Reserved							BSY	Reserved							DN
															rw								r								Rw

Figure 5.1:

- **Status Bits**: These bits indicate the current state of the core.
- **Interruption Flags:** These bits are used to generate an interruption request in the *int_req* signal of the AIP interface.
- **Mask Bits**: Each one of these bits can enable of disable the interruption flags.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
								Mask bits								**Status Bits**								**Interrupt/Clear Flags**							
Reserved								Reserved							MSK	Reserved							BSY	Reserved							DN
															rw								r								Rw

Figure 5.1 IP Convolutioner status register

Bits 31:24 – Reserved, must be kept cleared.

Bits 23:17 – Reserved Mask Bits for future use and must be kept cleared.

Bit 16 – **MSK:** mask bit for the DN (Done) interruption flag. If it is required to enable the DN interruption flag, this bit must be written to 1.

Bits 15:9 – Reserved Status Bits for future use and are read as 0.

Bit 8 – **BSY**: status bit "**Busy**".
Reading this bit indicates the current IP Convolutioner state:

0: The IP Convolutioner is not busy and ready to start a new process.
1: The IP Convolutioner is busy, and it is not available for a new process.

Bits 7:1 – Reserved Interrupt/clear flags for future use and must be kept cleared.

Bit 0 – **DN**: interrupt/clear flag "**Done**"
Reading this bit indicates if the IP Convolutioner has generated an interruption:

0: interruption not generated.

1: the IP Convolutioner has successfully finished its processing.

Writing this bit to 1 will clear the interruption flag DN.

5.2. Configuration delay register

Config: CConfigReg

Size: 32 bits

Mode: Write

This register is used to configure a delay time in milliseconds before the core starts copying data from the input to the output data memory. See Figure 5.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																	Data Y [14:5]										**Data X [4:0]**				
RESERVED																	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure 5.2 Configuration delay register.

Bits 31:15 – Reserved Data memory inputs. Must be kept cleared.

Bits 14:5 – **Data_X:** Data memory X input. Can be configured with a higher size.

Bits 4:0 – **Data_Y:** Data memory Y inputs. Must be kept cleared.

5.3. Input data memory (Mem X)

Config: MMemX

Size: 32 bits

Mode: Write

This memory is used to store data from an extern memory to be processed by the IP Convolutioner core. The size of this memory is set as a hardware parameter before the synthesis.

5.4. Output data memory (Mem Y)

Config: MMemY

Size: 32 bits

Mode: Write

This memory is used to store data from an extern memory to be processed by the IP Convolutioner core. The size of this memory is set as a hardware parameter before the synthesis.

5.5. Output data memory

Config: MMemOut

Size: 64 bits

Mode: Read

This memory is used to store processed data by the IP Convolutioner. After the IP Convolutioner completes its processing, the data stored in this memory will be result of the convolution stored by ***Mem X*** and ***Mem Y***. The size of this memory is set as a hardware parameter before the synthesis.

6. PYTHON DRIVER

The file *id1000500A.py* contains the convolutioner class definition. This class is used to control the IP Convolutioner core for python applications.

6.1. Usage example

In the following code a basic test of the IP Convolutioner core is presented. First, it is required to create an instance of the convolutioner object class. The constructor of this class requires the network address and port where the IP Convolutioner is connected, the communication port, and the path where the configs csv file is located. Thus, the communication with the IP Convolutioner will be ready. In this code, a function 'conv' is called. The function writes in both memories with random data by using the writeDataX and writeDataY methods. Then, the configurationRegister method is used to set the data in a single word to be processed, and then the startIP method is used to start core processing. Finally, the waitINT method is used to wait the activation of the DONE flag, and after that, the output data is read with the readData method.

```
import sys, random, time, os

logging.basicConfig(level=logging.DEBUG)
connector = '/dev/ttyACM0'
csv_file = '/home/ID1000500A_config.csv'
addr = 1
port = 0
aip_mem_size = 31

try:
    conv = convolutioner(connector, addr, port, csv_file)
    logging.info("Test Convolutioner: Driver created")
except:
    logging.error("Test Convolutioner: Driver not created")
    sys.exit()

random.seed(1)

X = [random.randrange(2**32) for i in range(0, aip_mem_size)]
Y = [random.randrange(2**32) for i in range(0, aip_mem_size)]

try:
    result = conv.conv(X,Y)
    logging.info(f"Convolution result: {[f'{x:08x}' for x in result]}")
except Exception as e:
    logging.error(f"Error in convolution {str(e)}")
conv.__finish__()
logging.info("The End")
```

6.2. Methods

6.2.1. Constructor

```
def __init__(self, connector, nic_addr, port, csv_file):
```

Creates an object to control the IP Convolutioner in the specified network address.

Parameters:

- connector (string): Communications port used by the host.
- nic_addr (int): Network address where the core is connected.
- port (int): Port where the core is connected.
- csv_file (string): IP Convolutioner csv file location.

6.2.2. writeDataX

```
def writeDataX(self, data):
```

Write data form memory X in the IP Convolutioner input memory ***MMemX***.

Parameters:

- data (List[int]): String array with the data to write.

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.3. writeDataY

```
def writeDataY(self, data):
```

Write data form memory Y in the IP Convolutioner input memory ***MMemY***.

Parameters:

- data (List[int]): String array with the data to write.

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.4. readData

```
def readData(self, size):
```

Read data from the IP Convolutioner output memory.

Parameters:

- size (int): Communications port used by the host.

Returns:

- List[int] Data read from the output memory.

6.2.5. startIP

```
def startIP(self):
```

Start processing in IP Convolutioner.

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.6. configurationRegister

```
def configurationRegister(self, dataSizeX, dataSizeY):
```

Set a string array assigned from writeDataX and writeDataY methods in IP Convolutioner processing.

Parameters:

- dataSizeX Assigned data from MemX.
- dataSizeY Assigned data from MemY.

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.7. enableINT

```
def enableINT(self):
```

Enable IP Convolutioner interruptions (bit DONE of the STATUS register).

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.8. disableINT

```
def disableINT(self):
```

Disable IP Convolutioner interruptions (bit DONE of the STATUS register).

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.9. status

```
def status(self):
```

Show IP Convolutioner status.

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.10. waitINT

```
def waitINT(self):
```

Wait for the completion of the process.

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.11. conv

```
def conv(self, X, Y):
```

The IP Convolutioner's functionality core. Conducts the convolution process on the input data X and Y.

Parameters:

- X Input data from Mem X.
- Y Input data from Mem Y.

Returns:

- conv_res(List[int]) Convolution process result.

7. C DRIVER

In order to use the C driver, it is required to use the files: *id1000500A.h, id01000500A.c* that contain the driver functions definition and implementation. The functions defined in this library are used to control the IP Convolutioner core for C applications.

7.1. Usage example

In the following code a basic test of the IP Convolutioner core is presented.

```
CODE SAMPLE:
const uint8_t nic_addr = 1;
    const uint8_t port = 0;
    const uint8_t AIP_MEM_SIZE = 31; //Size of the input and output memories
    const uint8_t sizeX = 5;
    const uint8_t sizeY = 10;

    ID1000500A_init("/dev/ttyACM0", nic_addr, port, "/home/ID1000500A_config.csv");

    srand(time(NULL));

    uint8_t X[AIP_MEM_SIZE];
    uint8_t Y[AIP_MEM_SIZE];

    printf("\nData generated with %i\n",AIP_MEM_SIZE);

    printf("\nX Data\n");

    for(uint8_t i=0; i<AIP_MEM_SIZE; i++){
        X[i] = rand() %0XFFFFFFFF;
        printf("%X\n", X[i]);
```

```
    }

    printf("\nY Data\n");
    for(uint8_t i=0; i<AIP_MEM_SIZE; i++){
        Y[i] = rand() %0XFFFFFFFF;
        printf("%X\n", Y[i]);

    }

    uint8_t sizeResult = sizeX + sizeY - 1;
    uint16_t result_conv[sizeResult];

    ID1000500A_conv(X, sizeX, Y, sizeY, result_conv);

    printf("\n\n Done detected \n\n");
    ID1000500A_status();

    printf("\n");
    for(int i=0; i<sizeResult; i++){
        printf("Rx[%d]: %X\n", i, result_conv[i]);
    }

    ID1000500A_finish();
```

7.2. Driver functions

7.2.1. id1000500A_init

```
int32_t id1000500A_init(const char *connector, uint_8 nic_addr, uint_8 port,
const char *csv_file)
```

Configure and initialize the connection to control the IP Convolutioner in the specified network address.

Parameters:

- connector: Communications port used by the host.
- nic_addr: Network address where the core is connected.
- port: Port where the core is connected.
- csv_file: IP Convolutioner csv file location.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.2. id1000500A_writeData

```
int32_t id1000500A_writeData(uint32_t* dataX, uint8_t sizeX, uint32_t* dataY,
uint_8 sizeY)
```

Write data in both IP Convolutioner input memories.

Parameters:

- dataX: Pointer to the first element to be written from Mem X.
- sizeX: Number of elements to be written for Mem Y.
- dataY: Pointer to the first element to be written from Mem Y.

- sizeY: Number of elements to be written for Mem Y.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.3. id1000500A_readData

```
int32_t id1000500A_readData(uint32_t *data, uint32_t data_size)
```

Read data from the IP Convolutioner output memory.

Parameters:

- data: Pointer to the first element where the read data will be stored.
- data_size: Number of elements to be read.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.4. id1000500A_startIP

```
int32_t id1000500A_startIP ()
```

Start processing in IP Convolutioner.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.5. id1000500A_configurationRegister

```
int32_t id1000500A_enableDelay(uint8_t sizeX, uint8_t sizeY)
```

Set a string array assigned from id1000500A_writeData in IP Convolutioner processing.

Parameters:

- sizeX: Assigned data from Mem X.
- sizeY: Assigned data from Mem Y.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.6. id1000500A_enableINT

```
int32_t id1000500A_enableINT()
```

Enable IP Convolutioner interruptions (bit DONE of the STATUS register).

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.7. id1000500A_disableINT

```
int32_t id1000500A_disableINT()
```

Disable IP Convolutioner interruptions (bit DONE of the STATUS register).

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.8. id1000500A_status

```
int32_t id1000500A_status()
```

Show IP Convolutioner status.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.9. id1000500A_waitINT

```
int32_t id1000500A_waitINT()
```

Wait for the completion of the process.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.10. id1000500A_finish

```
int32_t id1000500A_finish()
```

Finish IP Convolution connection.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.11. id1000500A_clearStatus

```
int32_t id1000500A_clearStatus()
```

Clear the status register.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.12. id1000500A_conv

```
int32_t id1000500A_conv (uint8_t* X, uint8_t sizeX, uint8_t* Y, uint8_t sizeY,
uint16_t* result)
```

Main functionality core. Conducts the convolution process.

Parameters:

- dataX: Pointer to the first element to be written from Mem X.
- sizeX: Number of elements to be written for Mem Y.
- dataY: Pointer to the first element to be written from Mem Y.
- sizeY: Number of elements to be written for Mem Y.
- result: Convolution process result.

Returns:

- int16_t Return result whether the function has been completed successfully.