VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Computer Architecture Assignment (CO2008)

REPORT
CONVOLUTION OPERATION

Instructor: Nguyen Thien An
Students: Hua Vu Minh Hieu - 2052990

HO CHI MINH CITY, NOVEMBER 2024

Contents

1 Introduction **2**

2 System design overall **3**

3 Methodology **4**
3.1 How I read file and process the character . . . 4
3.2 Apply padding to image matrix . . . 7
3.3 Perform convolutional operation . . . 8
3.4 Rounding processing . . . 10
3.5 Write to file processing . . . 11

4 Implementation **14**
4.1 Reading N, M, P, S value from input file . . . 14
4.2 Reading the value of matrix and store to memory . . . 16
4.3 Apply padding to image matrix . . . 18
4.4 Perform convolution operation and dot product function . . . 21
4.5 Process the output matrix to buffer of character function . . . 25
4.5.1 is_increase_integer_part function . . . 28
4.5.2 extract_integer_part function . . . 30
4.5.3 extract_decimal_part function . . . 31
4.5.4 reverse_buffer function . . . 33
4.5.5 write_digit_to_buffer function . . . 35
4.6 write_to_file function . . . 36

5 Result **39**
5.1 Testcase 1 . . . 39
5.2 Testcase 2 . . . 40
5.3 Testcase 3 . . . 41
5.4 Testcase 4 . . . 42
5.5 Testcase 5 . . . 43
5.6 Testcase with input error . . . 44

1 Introduction

Convolution is a fundamental operation in computer science, particularly in fields such as image processing, computer vision, and deep learning. It forms the backbone of convolutional neural networks (CNNs), which are widely used for analyzing visual data. By applying a kernel (a small matrix of weights) over an input image, convolution extracts essential features such as edges, textures, and patterns. This operation enables machines to recognize and interpret visual information, making it integral to applications like facial recognition, medical imaging, and autonomous vehicles. Understanding the underlying principles of convolution is crucial for building efficient and effective algorithms in these domains.

This assignment focused on implementing the convolution operation using the MIPS assembly language. The tasks included reading a matrix and kernel from an external input file, applying specified parameters such as padding and stride, and computing the convolution to generate an output matrix. The program was designed to handle floating-point numbers, apply symmetrical zero-padding, and ensure accuracy in the convolution process. Additionally, the implementation adhered to predefined constraints, including memory allocation for matrices and using MIPS-specific instructions for data manipulation and arithmetic operations. The result was saved to an output file, applying transform between character and floating-point number.

2 System design overall

The system for performing the convolution operation is designed to follow a structured and logical workflow, ensuring accurate results and proper error handling. It begins with reading an input file containing essential parameters and matrices. The program validates the file to ensure it adheres to the required format. If the file is invalid, an error is logged, and the program is terminated.

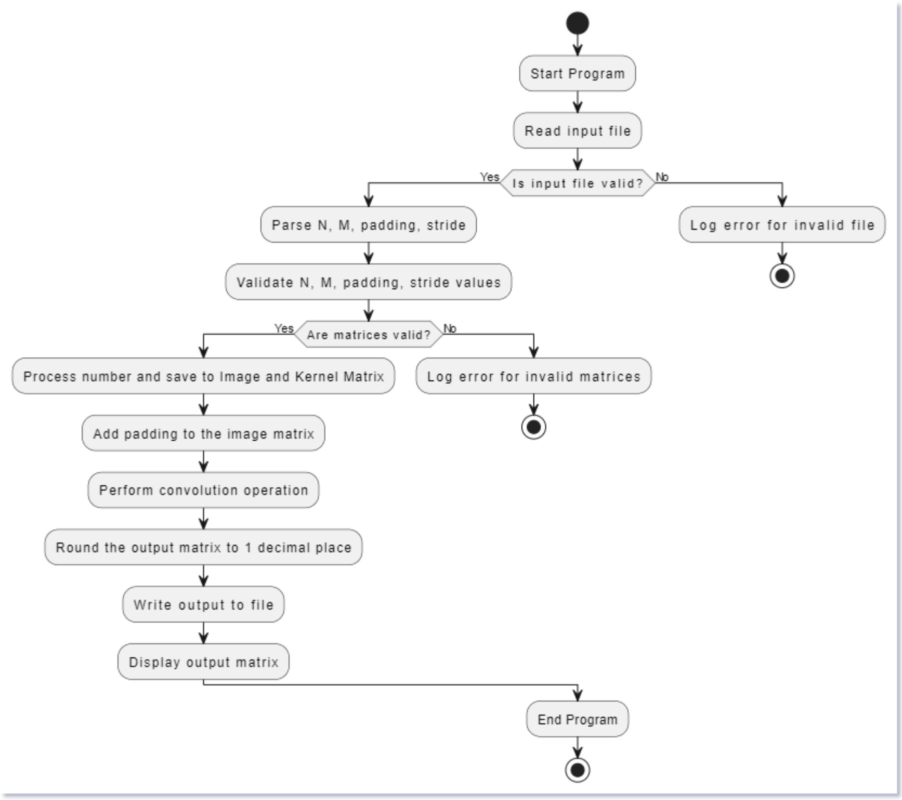

Figure 1: *The flow of the whole program*

Once the file is verified, the program extracts the matrix dimensions, padding, and stride values, and validates their correctness. If the matrices or parameters are invalid, appropriate errors are logged, and execution stops. Valid inputs are processed and stored in memory, with padding applied to the image matrix as specified.

The core of the system involves performing the convolution operation, where the kernel matrix slides over the input matrix, and element-wise calculations are performed to produce the output matrix. The output values are rounded to one decimal place to maintain precision and consistency. Finally, the result is written to an output file and displayed for the user, marking the end of the program.

3 Methodology

3.1 How I read file and process the character

Firstly, I will read the value N, M, P, S. The flow chart below describe the flow how i do that:

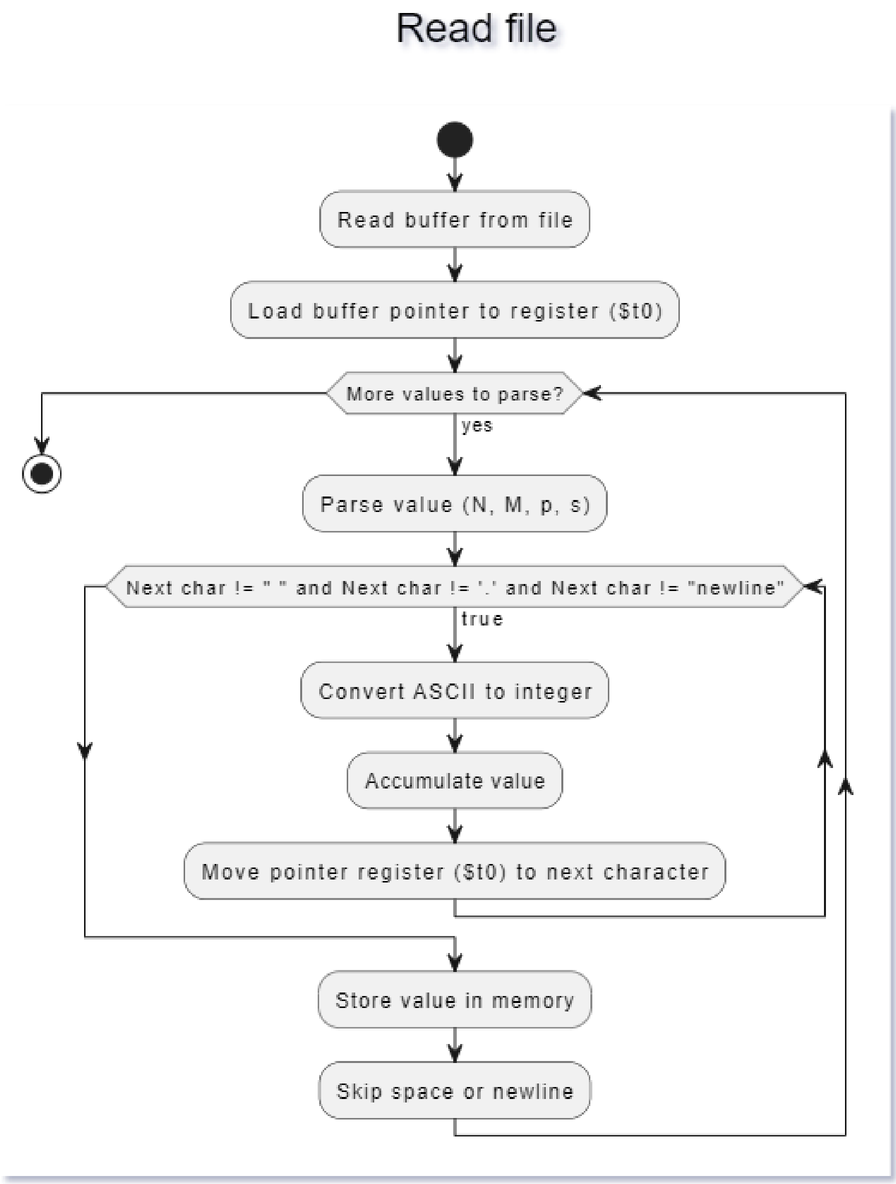

Figure 2: *The flow chart of processing N, M, P, S from character to integer*

The flowchart illustrates the step-by-step process of reading and processing the values of N, M, p, and s from a buffer file. The system begins by loading the entire file buffer into memory and initializing a pointer to the starting position of the buffer (stored in register $t0).

- **Check for Remaining Values:** The program checks if there are more values (N, M, p, s) left to parse. If no more values are available, the program terminates this step.

- **Parsing Each Value:** If values remain, the pointer iterates through the buffer to extract each value. Each value is parsed as a sequence of characters from the buffer.
- **Character-by-Character Processing:** It converts ASCII characters into their corresponding integer representation then accumulates these integers to form the complete numeric value. Finally, moves the pointer to the next character for further processing.
- **Handling Delimiters:** The program continues parsing characters until it encounters a space (" "), period ("."), or newline character ("\n"), indicating the end of the current value.
- **Storing the Value:** Once a complete value is formed, it is stored in memory for further use. The program then skips any remaining delimiters or whitespace to position the pointer for the next value.

The process repeats for all four values (N, M, p, s) in sequence, ensuring each is correctly parsed, converted, and stored.

After successfully reading the values N, M, p, and s from the input buffer, the next step is to ensure that these values meet the specified constraints. This process is depicted in the flowchart and consists of two main stages: general input validation and kernel size validation.

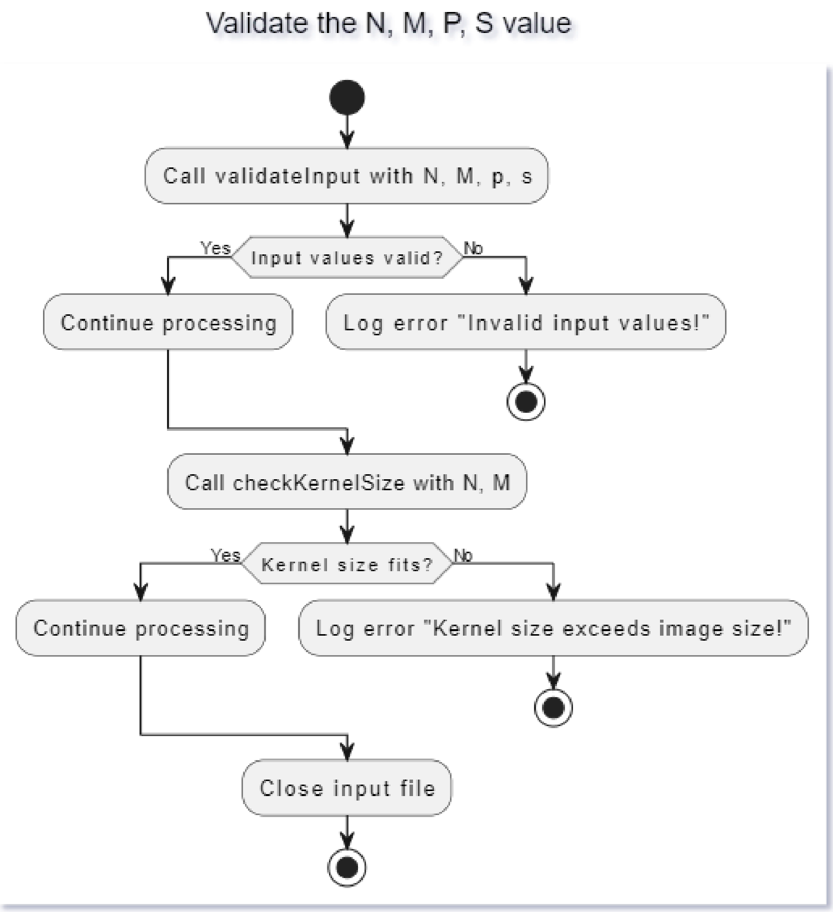

Figure 3: *Flow chart of validate the input*

The program calls the **validateInput function** with the parameters N, M, p, and s. This function checks if the values are within the required ranges:

- N (size of the image matrix) should be between 3 and 7
- M (size of the kernel matrix) should be between 2 and 4.
- p (padding) should be between 0 and 4.
- s (stride) should be between 1 and 3.

If any value is outside its allowed range, an error message is logged ("Invalid input values!") and the program halts further processing.

If the general input validation passes, the program proceeds to call the **checkKernel-Size** function with the parameters N and M. This function ensures that the kernel size does not exceed the dimensions of the image matrix $M \leq N$. If the kernel size is invalid $M > 0$, an error is logged ("Kernel size exceeds image size!") and the program stops.

If both checks pass, the program continues reading and processing the value of image and kernel matrix. The process begins by loading the dimensions of the image matrix (N) and the kernel matrix (M). Using these values, the total memory required for both matrices is calculated as [(N*N) + (M*M)] * 4 bytes, assuming each element is a 4-byte floating-point number.

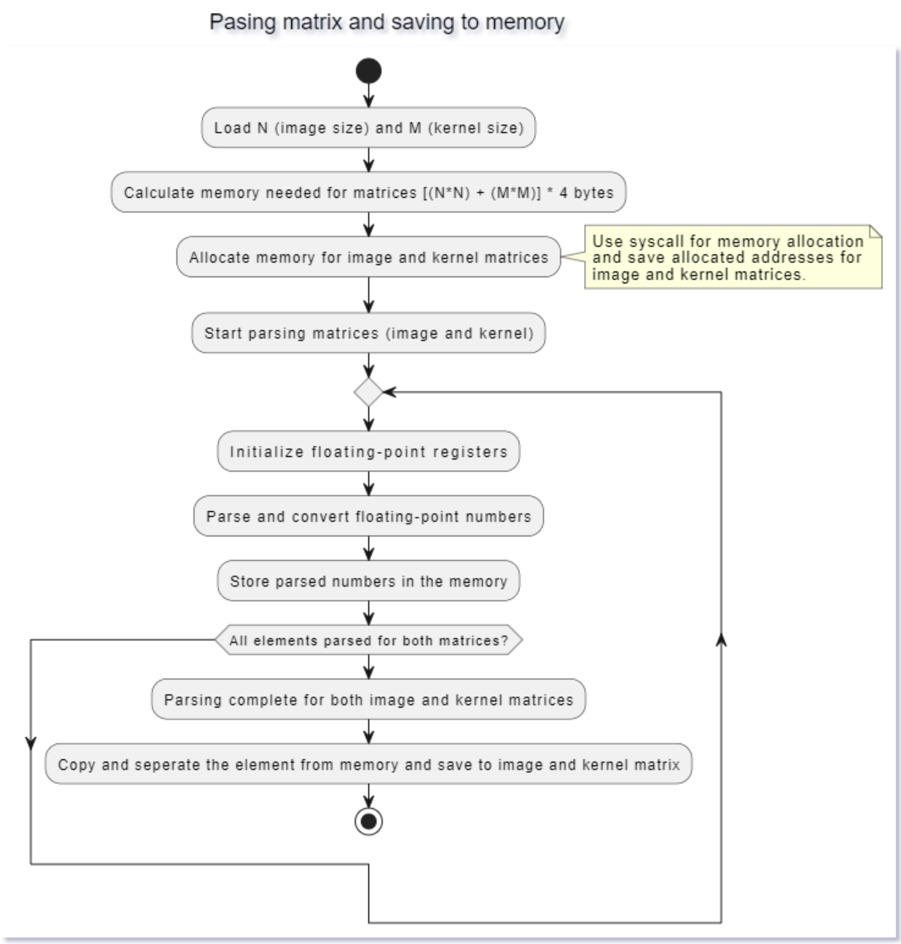

Figure 4: *The flow chart decribe the process of parsing element of image and kernel matrix*

For each element:

- Floating-point registers are set up to handle the parsing and conversion of values.
- The input values are parsed from the data source (e.g., a file or buffer) and converted into floating-point numbers.
- Each parsed value is stored in the allocated memory for the respective matrix.

The process continues until all elements of both matrices are parsed. A condition checks whether all N*N elements of the image matrix and all M*M elements of the kernel matrix have been successfully processed. Once parsing is complete, the program separates the elements in memory and saves them explicitly to the respective locations for the image and kernel matrices.

3.2 Apply padding to image matrix

The process begins by loading the size of the image matrix (N) and the padding value (p). These parameters define the original matrix size and the number of rows and columns of padding to add around the matrix.

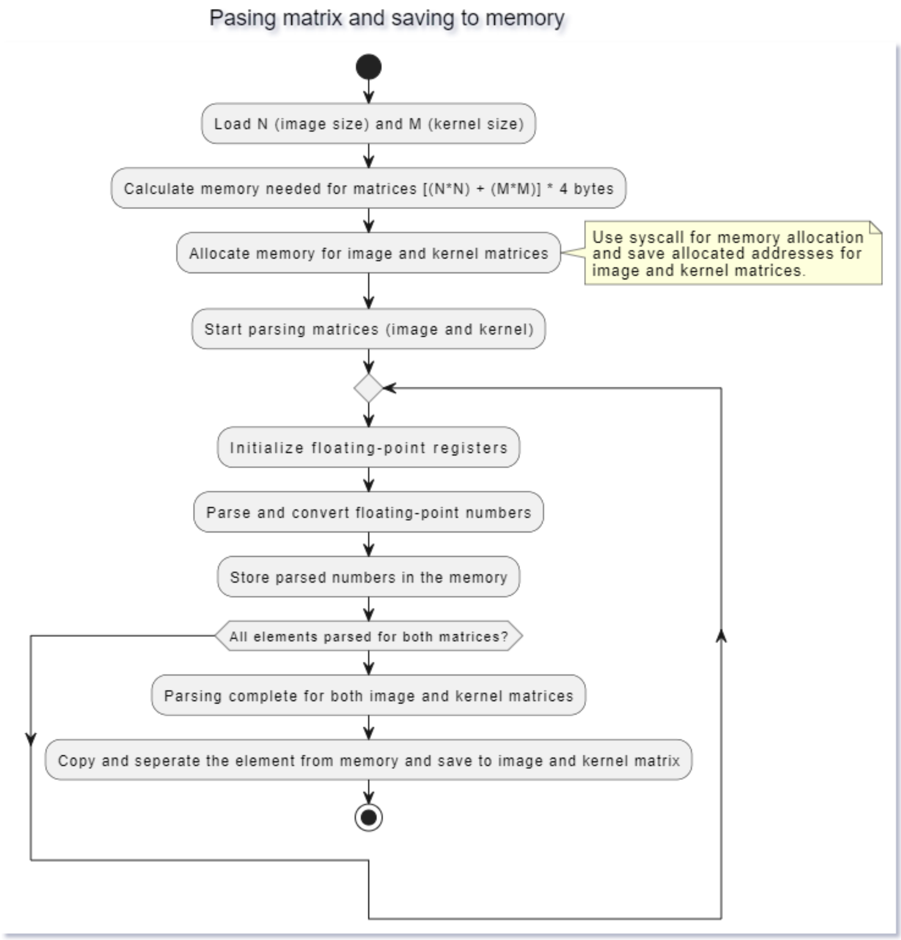

Figure 5: *The flow chart describe the process of apply padding to image matrix*

The dimensions of the padded matrix are calculated as (N + 2p) x (N + 2p), where p is the padding size. This accounts for p rows and columns of zeros added to all sides of the original matrix.

Memory is allocated for the new, larger matrix to store the padded values. The allocated size corresponds to the dimensions calculated in the previous step. A syscall is used to handle memory allocation, and the address of the padded matrix is saved for further processing.

The padded matrix is initialized with zeros to represent the padding. This ensures that all padding regions are filled with zero values, as required by the padding technique.

The elements of the original image matrix are copied into the central region of the padded matrix, surrounded by the zero-padding rows and columns. After copying the original matrix into the padded structure, the process of padding the matrix is complete. The padded matrix is now ready for convolution operations.

3.3 Perform convolutional operation

To perform the convolution operation, I used the **row-major** addressing formula to access elements in the image and kernel matrices stored in memory. The formula calculates the memory address of a specific element based on its row and column indices:

$$\mathbf{addr} = \mathbf{baseAddress} + (\mathbf{rowIndex} \times \mathbf{colSize} + \mathbf{colIndex}) \times \mathbf{dataSize}$$

Explanation of the Formula:

- **baseAddress:** The starting memory address of the matrix.
- **rowIndex:** The index of the desired row (0-indexed).
- **colIndex:** The index of the desired column (0-indexed).
- **colSize:** The total number of columns in the matrix.
- **dataSize:** The size of each matrix element in bytes (4 bytes for floating-point numbers).

Using this formula, I accessed elements of the image and kernel matrices while performing the convolution operation.

The convolution process involves sliding the kernel matrix over the image matrix (padded matrix). For each position of the kernel, I calculated the dot product of the overlapping elements. The row-major formula was used to access the required elements efficiently.

To access an element of the image matrix at position $(i + x, j + y)$, where (i, j) is the current top-left corner of the kernel in the image, and (x, y) is the relative position of the element within the kernel, the address was calculated as:

$$\mathbf{addr_{image}} = \mathbf{imageBaseAddr} + ((i + x) \times \mathbf{colSize} + (j + y)) \times \mathbf{dataSize}$$

Similarly, to access an element of the kernel matrix at position (x, y), the address was calculated as:

$$\mathbf{addr_{kernel}} = \mathbf{kernelBaseAddr} + (x \times \mathbf{kernelColSize} + y) \times \mathbf{dataSize}$$

The result of the dot product was stored in the output matrix. The address of an element in the output matrix at position (i, j) was computed as:

$$\mathbf{addr_{output}} = \mathbf{outputBaseAddr} + (i \times \mathbf{outputColSize} + j) \times \mathbf{dataSize}$$

This flow chart below describe the process of I how I performed the convolution operation: To begin, the program loads the memory addresses of the image matrix, kernel

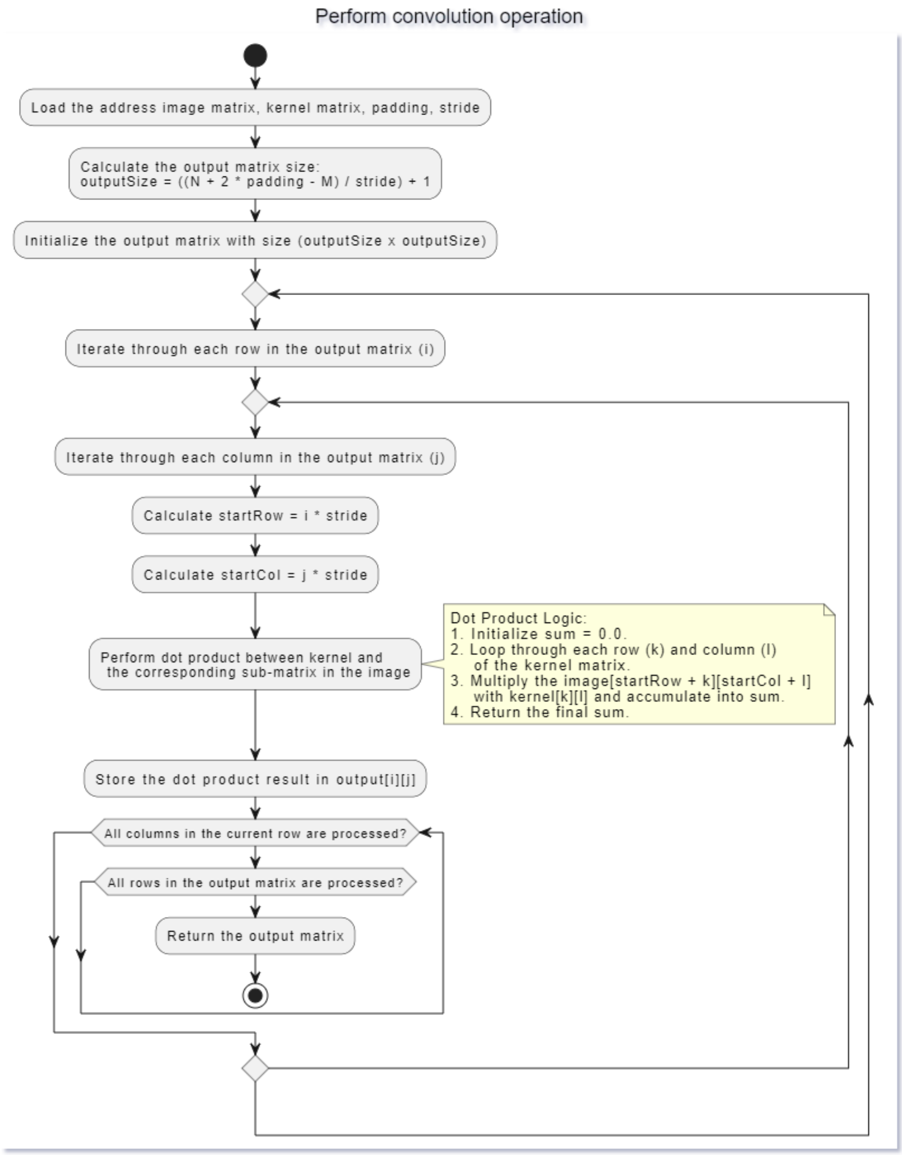

Figure 6: *The process of performing convolution opeartion*

matrix, padding, stride, and the original size of the image matrix N). Using these parameters, the size of the output matrix is calculated with the following formula:

$$\text{outputSize} = \left(\frac{N + 2 \times \text{padding} - M}{\text{stride}}\right) + 1$$

The program iterates through every element of the output matrix to compute the convolution result.

- **Row Iteration**: For each row (i) in the output matrix:
  - The starting row in the image matrix is calculated as:

$$\text{startRow} = i \times \text{stride}$$

- **Column Iteration**: For each column (j) in the output matrix:
  - The starting column in the image matrix is calculated as:

$$\text{startCol} = j \times \text{stride}$$

At each position (i, j) in the output matrix, the program calculates the dot product between the kernel matrix and the corresponding sub-matrix of the image matrix.

At each position (i, j) in the output matrix, the program calculates the dot product between the kernel matrix and the corresponding sub-matrix of the image matrix.

- **Logic for Dot Product**:
  1. Initialize sum = 0.0.
  2. Loop through each element in the kernel matrix, with indices (k, l), where k is the row index and l is the column index:
     - Access the corresponding element in the image matrix at position $(\text{startRow} + k, \text{startCol} + l)$.
     - Multiply the image matrix element with the kernel element and accumulate the result:

$$\text{sum}+ = \text{image}[\text{startRow} + k][\text{startCol} + l] \times \text{kernel}[k][l]$$

- **Store the Result**:
  - Once the dot product computation is complete, store the result in the output matrix at position (i, j):

$$\text{output}[i][j] = \text{sum}$$

Once all rows and columns in the output matrix are computed, the program returns the output matrix containing the convolution results

3.4 Rounding processing

In the assignment, it is required to round floating-point numbers to one decimal place to ensure consistent precision in output. The process involves isolating and analyzing the decimal part of a number, determining whether to round it up or not based on the second decimal digit, and then storing the rounded result.

The program begins by receiving the floating-point number stored in register $f12. The integer part of the number is extracted by converting the floating-point value to an integer then subtract the integer part from the original floating-point number to isolate the decimal portion.

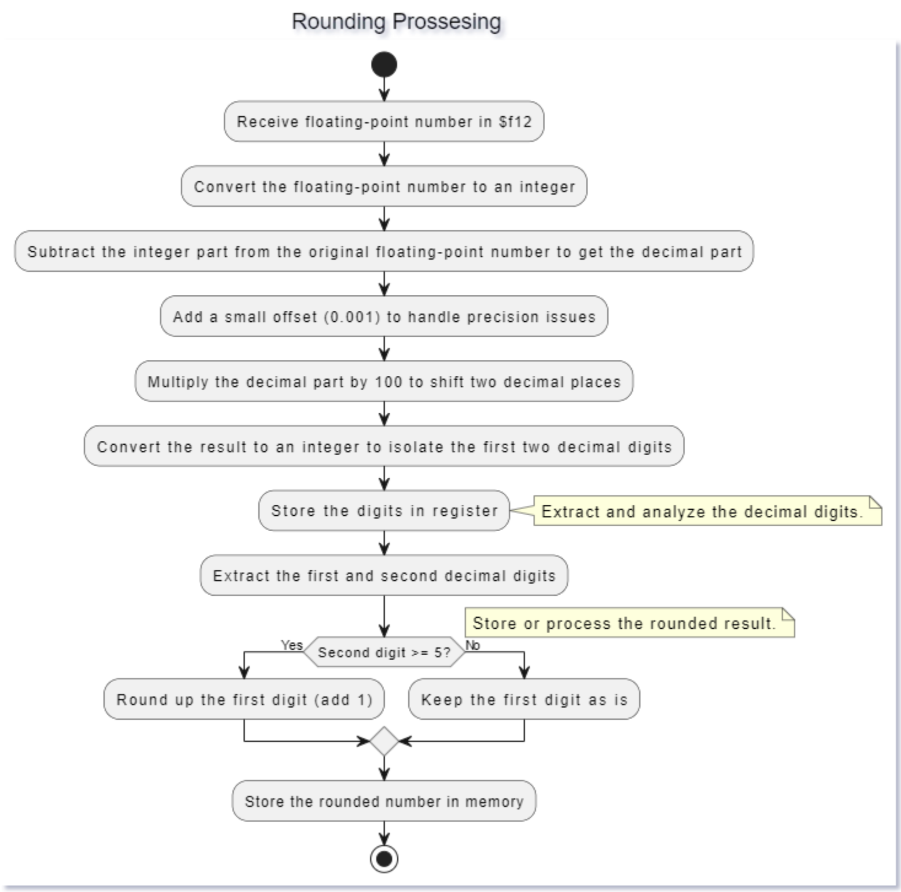

Figure 7: *The flow chart describe the process of rounding floating-point number*

To address potential floating-point precision errors, I added small offset with value 0.001 to the decimal part. The adjusted decimal part is multiplied by 100 to shift the first and second decimal digits into the integer range and then converted to an integer to isolate the first two decimal digits

If the second decimal digit is greater than or equal to 5, the first decimal digit is incremented by 1 to round up.

If the second decimal digit is less than 5, the first decimal digit remains unchanged.

3.5 Write to file processing

Before writing the output matrix to **output.txt** file. I need to process each element of output matrix, extract each value from number to character and store it to a **bufferOutput**

This flow chart below will describe this process:

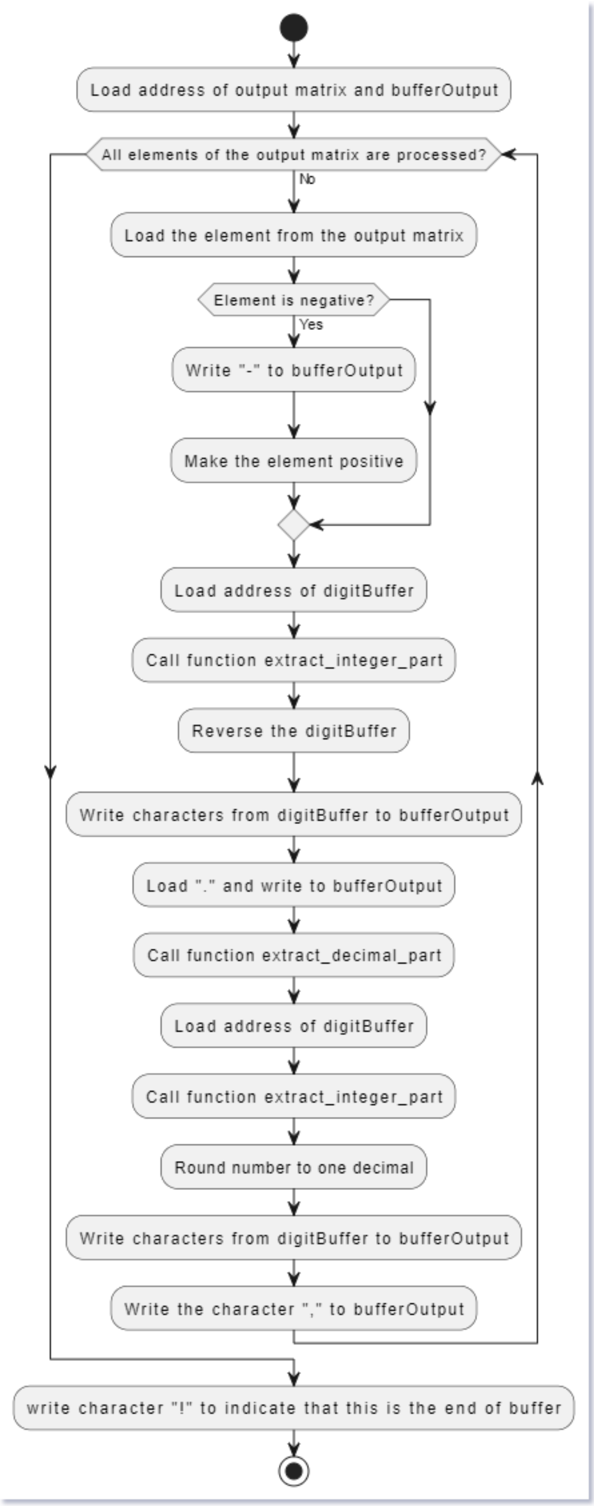

Figure 8: *The flow chart describe how to process element and then store in bufferOutput*

I will give an example to describe this process:
Example: **[-123.456, 52.43,4.3]**
bufferOutput: []
digitBuffer: []

- **Step 1:** Load the element **-123.456**

- **Step 2:** Check element is positive or negative. The number is negative
  -> write **"-"** to buffer
  -> bufferOutput: **[-]**

- **Step 3:** Call the function **extract_integer_part**:
  Integer part = **123**
  Store integer part in digiBuffer **[321]**
  Reverse the digitBuffer to get **123** -> **digitBuffer:[123]**
  Write **123** to **bufferOutput**
  -> **bufferOutput: [-123]**

- **Step 4:** Write **"."** to bufferOutput
  -> **bufferOutput: [-123.]**

- **Step 5:** Extract the Decimal Part and Rounding Number:
  Decimal part = **0.456**
  Call function: **extract_decimal_part** to get **456**
  Call **round_up** Function to round the decimal part: **0.456** -> **0.5**
  Extract digits 5 into digitBuffer: **[5]**
  Write **5** to bufferOutput -> **bufferOutput: [-123.5]**

- **Step 6:** Write **","** to bufferOutput: -> **bufferOutput: [-123.5,]**
  ...
  ...
  If the final element is processed. I will write the character **"!"** to **bufferOutput**. This character will be as an indicator that help me to know this is the end of **bufferOuput** when write the character to the **output.txt** file.

  Perform the same process with the next element. The **bufferOutput** will have the result as follow:

$$[-123.5, 52.4, 4.3!]$$

After this process successfully, every things behinds seem to be easy. Now I just opened the output file. Load each character from the buffer to register and write it to the output.txt file.

One thing to noticed that I replaced the character **","** from bufferOutput to character **" " (space)** when performing write to file

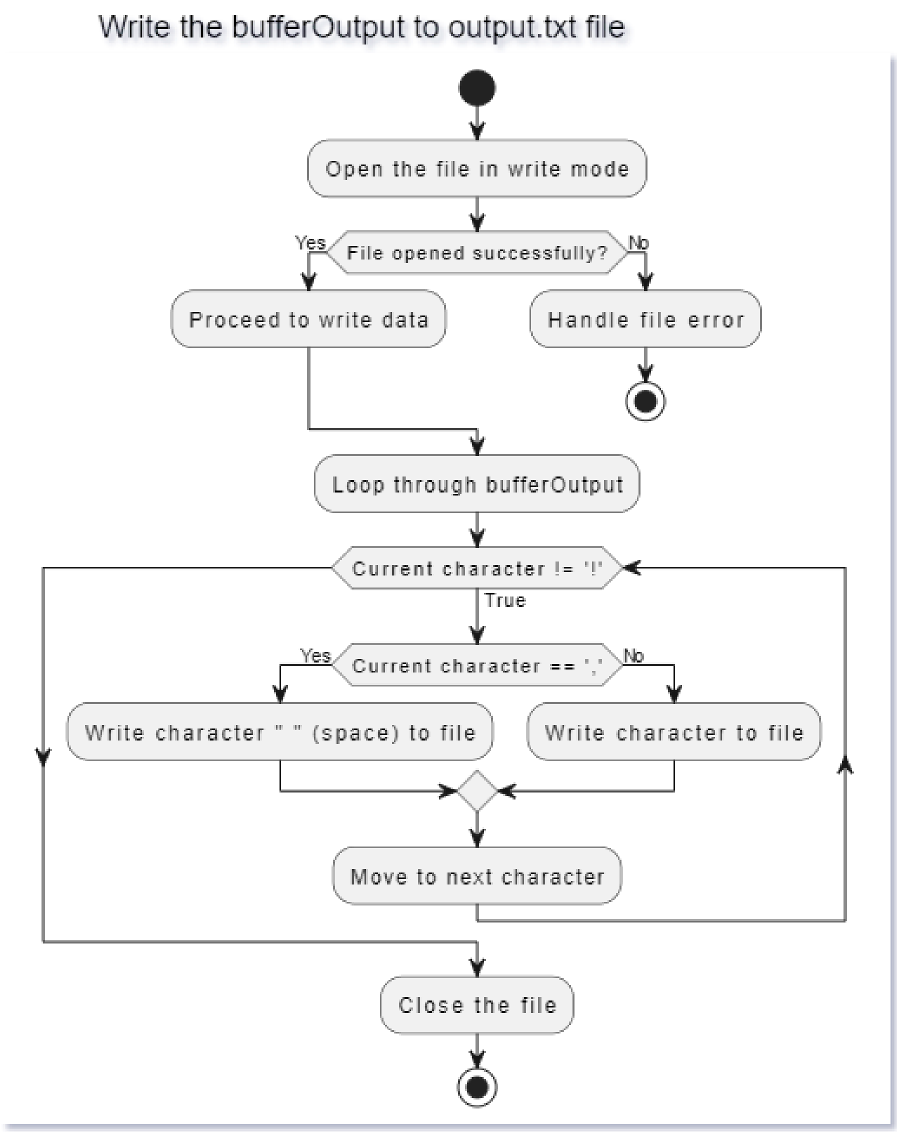

Figure 9: *The flow chart describe the process of writing character from bufferOutput to output.txt*

4 Implementation

In this section, I will describe the key functions and logical processes implemented to perform the convolution operation.

The following subsections outline the critical steps and functions implemented for this assignment. These include reading input matrices, applying padding, performing the convolution operation, and writing the results to an output file. Each function is explained with its logic, purpose, and role in the overall system.

4.1 Reading N, M, P, S value from input file

```
    la $t0, buffer
    # --- Parsing N ---
    li $t1, 0 #register to store parsed value of N, M, P, S
parse_N_loop:
    lb $t2, 0($t0)          # Load byte
```

```
     # If the character is "." skip to next M value
    beq $t2, 46, parse_M
    # Convert ASCII to integer (subtract ASCII '0')
    sub $t2, $t2, 48
    # Shift left
    mul $t1, $t1, 10
    # Add the parsed digit to the value
    add $t1, $t1, $t2
    addi $t0, $t0, 1          # Move to the next character
    j parse_N_loop            # Continue parsing N
parse_M:
    sw $t1, N                 # Store parsed N
    # --- Skip space after N ---
    addi $t0, $t0, 3          # Skip the space after N

    # --- Parsing M ---
    li $t1, 0                 # Reset the value for M
parse_M_loop:
    lb $t2, 0($t0)            # Load the next character for M
    beq $t2, 46, parse_p      # Break on space to parse next integer (padding)
    sub $t2, $t2, 48          # Convert ASCII to integer
    mul $t1, $t1, 10          # Multiply by 10 to shift left
    add $t1, $t1, $t2         # Add parsed digit to M
    addi $t0, $t0, 1          # Move to the next character in the buffer
    j parse_M_loop            # Loop until the space
parse_p:
    sw $t1, M                 # Store parsed M

    # --- Skip space after M ---
    addi $t0, $t0, 3          # Skip the space after M

    # --- Parsing p (padding) ---
    li $t1, 0                 # Reset the value for p
parse_p_loop:
    lb $t2, 0($t0)            # Load the next character for p
    beq $t2, 46, parse_s
    sub $t2, $t2, 48          # Convert ASCII to integer
    mul $t1, $t1, 10          # Multiply by 10 to shift left
    add $t1, $t1, $t2         # Add parsed digit to p
    addi $t0, $t0, 1          # Move to the next character
    j parse_p_loop            # Loop until the space
parse_s:
    sw $t1, p                 # Store parsed p

    # --- Skip space after p ---
```

```
    addi $t0, $t0, 3        # Skip the space after p

    # --- Parsing s (stride) ---
    li $t1, 0               # Reset the value for s
parse_s_loop:
    lb $t2, 0($t0)          # Load the next character for s
    beq $t2, 46, done_parsing
    sub $t2, $t2, 48        # Convert ASCII to integer
    mul $t1, $t1, 10        # Multiply by 10 to shift left
    add $t1, $t1, $t2       # Add parsed digit to s
    addi $t0, $t0, 1        # Move to the next character
    j parse_s_loop          # Loop until newline
done_parsing:
    sw $t1, s               # Store parsed s
```

4.2 Reading the value of matrix and store to memory

```
     # --- Parsing the image matrix ---
    # Move pointer after the first row
    addi $t0, $t0, 4
    li $t3, 0     #variable for check negative
    li $t5, 0       #counter for image matrix elements
parse_matrix:
    # Stop when all elements are parsed
    beq $t5, $t6, done_parsing_matrices

    # Initialize the floating-point registers properly using
   integer registers and mtc1
    li $t1, 0x00000000      # Load 0 into integer register
    mtc1 $t1, $f0           # register $f0 -> accumulator
    mtc1 $t1, $f6           # register $f6 -> fractional
   accumulator
    li $t1, 0x41200000      # Load 10.0  into $t1
    mtc1 $t1, $f9           # register $f9 -> divisor for
   fractional part

    jal parse_floating_point   # Call function to parse a
   floating-point number
    s.s $f0, 0($t7)            # Store parsed floating-point
   number into image matrix
    addi $t7, $t7, 4           # Move to next element
    addi $t5, $t5, 1           # Increment the index
    j parse_matrix
done_parsing_matrices:
    jal apply_padding_to_iamge_matrix
    # --- Function to parse a floating-point number from the
```

```
   buffer ---
parse_floating_point:
    lb $t2, 0($t0)
    beq $t2, 45, saveSign
    beq $t2, 0, finish_parsing              # If null
   terminator (end of string), stop
    beq $t2, 46, parse_fraction_part        # If '.', switch
   to parsing the fractional part
    beq $t2, 32,  finish_parsing            # If space, we're
    done with this number
    beq $t2, 10, finish_parsing
    beq $t2, 13, finish_parsing

    #parse integer part
    sub $t2,$t2,48 #convert to integer
    mtc1 $t2,$f2   #move integer into floating point number
    cvt.s.w $f2,$f2 #convert integer to single-precision
   float

    mul.s $f0,$f0,$f9 #multiply accumlator by 10 (shift left)
    add.s $f0,$f0,$f2 #add the current digit to the
   accumulator

    add $t0,$t0,1 #move to the next charater
    j parse_floating_point
nextDigit:
  addi $t0,$t0,1
  lb $t2, 0($t0) #if the next byte is not "." => does not
   have fraction part
  beq $t2, 32, finish_parsing
  j parse_floating_point
saveSign:
  addi $t3,$t3, 1    #the is number is negative number
  addi $t0, $t0, 1 #ignore this sign
  j parse_floating_point

parse_fraction_part:
  addi $t0,$t0,1

parse_fraction_loop:
  lb $t2, 0($t0)
  beq $t2,0, finish_parsing
  beq $t2,32, finish_parsing    #if space, we are with this
   number
  beq $t2,10, finish_parsing    #if newline, we are done with
    this array
  beq $t2,13, parse_kernel_matrix    #jump to parse the
```

```
 kernel matrix
  sub $t2,$t2, 48
  mtc1 $t2,$f2
  cvt.s.w $f2,$f2
  div.s $f2, $f2, $f9     # divide the digit by the dividor
  add.s $f6,$f6, $f2      # accumlate the fractional part
  li $t1, 0x41200000      # Load 10.0 (IEEE 754 for 10.0) into
   $t1
    mtc1 $t1, $f10          # Move 10.0 into floating-point
 register $f9 (divisor for fractional part)
  mul.s $f9,$f9,$f10      #multiply divisor by 10 for the next
   fractional digit
  addi $t0,$t0,1
  j parse_fraction_loop

finish_parsing:
  addi $t0,$t0,1
  add.s $f0,$f0,$f6 #add fractional part to the integer part
  beq $t3, 1, conToNegNum
  jr $ra

parse_kernel_matrix:
  addi $t0,$t0,2
  add.s $f0,$f0,$f6 #add fractional part to the integer part
  beq $t3, 1, conToNegNum
  jr $ra

conToNegNum:
  neg.s $f0, $f0
  li $t3, 0
  jr $ra
```

4.3 Apply padding to image matrix

```
    ##### INITIALIZE THE PADDING IMAGE #####
initialize_padded_image:
    lw  $t2, p                  # Load the padding size
    lw  $t3, N                  # Load the image matrix size
    (N)
    # Calculate the size of the padded matrix (N + 2 *
  padding)
    li  $t4, 2
    mul $t1, $t2, $t4           # t1 = 2 * padding
    add $t1, $t1, $t3           # t1 = N + 2 * padding (size
    of the padded matrix)
    sw $t1, paddedSize
```

```
    # Calculate total bytes needed for the padded matrix
    li  $t4, 4              # Each element is 4 bytes (float)
    mul $t2, $t1,$t1    # Total element for padded matrix
    mul $t3, $t2, $t4       # t2 = total bytes needed for the matrix

    move $a0, $t3           # Number of bytes to allocate
    li   $v0, 9             # Syscall for memory allocation
    syscall                 # Allocate memory
    move $s2, $v0           # Save the allocated base address of the padded matrix in $s2
    move $t4, $s2           #$t2 is the base address of the padded image -> use for moving

    ##### INITIALIZE WITH 0.0 #####
    la  $t0, zero           # Load the address of 0.0
    lwc1 $f0, 0($t0)        # Load 0.0 into the floating-point register $f0

    li  $t5, 0              # Initialize counter for the number of elements
initialize_loop:
    bge $t5, $t2, initialize_done # Exit loop if all elements initialized
    # Store 0.0 into the current matrix position
    swc1 $f0, 0($t4)                # Store 0.0 into the padded matrix
    addi $t4, $t4, 4                # Move to the next float element
    addi $t5, $t5, 1                # Increment  counter
    j    initialize_loop

initialize_done:

    #### INSERT IMAGE INTO THE PADDING IMAGE ####
insert_image_to_padded:
    lw  $t0, p                      # Load the padding size
    lw  $t1, paddedSize             #size of padded matrix
    lw  $t8, N   # Load the image matrix size (N)
    la   $t2, image

    li   $t3, 0
insert_image_rows:
    beq  $t3, $t8, insert_done   # If all rows of the image matrix are copied, exit the loop
    li   $t4, 0                     # Initialize column index for the image matrix
```

```

insert_image_columns:
    bge  $t4, $t8, next_image_row # If all columns of the image matrix row are copied, move to the next row

    # Load the element from the image matrix
    l.s $f0, 0($t2)             # Load the element at position (row, col) of the image matrix
    add  $t5, $t3, $t0          # Calculate padded_row = image_row + paddingNum
    add  $t6, $t4, $t0          # Calculate padded_col = image_col + paddingNum

    # Calculate the address in the padded matrix:
    # = padded_base + ((padded_row * padded_size) + padded_col) * 4
    mul  $t7, $t5, $t1           # t7 = padded_row * padded_size
    add  $t7, $t7, $t6           # t7 = padded_row * padded_size + padded_col
    sll  $t7, $t7, 2             # t7 = (padded_row * padded_size + padded_col) * 4 (byte offset)

    # Now, calculate the actual address using the base of the padded matrix
    add  $t7, $s2, $t7           # t7 is the actual address of the padded matrix

    # Store the image matrix element into the calculated address in the padded matrix
    swc1 $f0, 0($t7)             # Store the element at the calculated position

    # Move to the next column in the image matrix
    addi $t2, $t2, 4             # Move to the next element
    addi $t4, $t4, 1             # Increment the column index
    j    insert_image_columns    # Repeat for the next column

next_image_row:
    addi $t3, $t3, 1             # Move to the next row
    j    insert_image_rows       # Repeat for the next row

insert_done:
    j perform_convolution_operation
```

4.4 Perform convolution operation and dot product function

```
  #### PERFORM CONVOLUTIONAL OPERATION ####

# Step 1: Load size of padded matrix, kernel size, padding, and stride
  la   $t0, N       # Load address of padded matrix size
  lw   $t1, 0($t0)  # Load size of padded matrix (N)

  la   $t2, M       # Load address of kernel matrix size
  lw   $t3, 0($t2)  # Load size of kernel matrix (M)

  la   $t4, p       # Load address of padding value
  lw   $t5, 0($t4)  # Load padding value (p)

  la   $t6, s       # Load address of stride value
  lw   $s5, 0($t6)  # Load stride value (s)

  addi  $s2, $s2, 0 #base address of padded matrix is $s2
  la  $s3, kernel

  # t8 = sizeOfPaddedMatrix - sizeOfKernelMatrix
  sub  $t8, $t1, $t3

  # t9 = 2 * paddingValue (left shift padding by 1)
  sll  $t9, $t5, 1

  # t8 = (sizeOfPaddedMatrix - sizeOfKernelMatrix) + 2 * paddingValue
  add  $t8, $t8, $t9
  div  $t8, $t8, $s5                # t8 = t8 / stride
  addi $t8, $t8, 1                  # Add 1 for the output size
  sw   $t8, outputSize              # Store output size (dimension)

  lw   $t9, outputSize
  # Calculate total elements: dimension * dimension
  mul  $t9, $t9, $t9
  li   $t0, 4                       # Each element is 4 bytes
  mul  $t9, $t9, $t0                # Total bytes to allocate: total elements * 4
  li   $v0, 9                       # System call for sbrk (memory allocation)
  move $a0, $t9                     # Request memory of size in bytes
```

```
    syscall
    move $s4, $v0                    # Save base address of dynamically allocated output matrix in $s4
    move $s7, $s4 #use for increase space
    # Start the convolution operation loop
    li   $t0, 0                      # i = 0 (initialize output row index)
conv_row_loop:
    lw   $t1, outputSize             # Load output matrix dimension
    bge  $t0, $t1, print_matrix_convol      # If i >= outputSize, exit loop

    li   $t2, 0                      # j = 0 (initialize output column index)
conv_col_loop:
    bge  $t2, $t1, nextRow          # If j >= outputSize, go to next row

    # Calculate starting row and column for this convolution step
    mul  $t3, $t0, $s5               # startRow = i * stride
    mul  $t4, $t2, $s5               # startCol = j * stride

    # Push necessary registers onto the stack
    addi $sp, $sp, -20               # Make space on the stack
    sw $t0, 0($sp)                   # Save $t0 (row index)
    sw $t1, 4($sp)    #save for the size of output
    sw $t2, 8($sp)                   # Save $t2 (column index)
    sw $t3, 12($sp)                   # Save $t3 (startRow)
    sw $t4, 16($sp)                  # Save $t4 (startCol)

    # Call dotProduct function
    move $a0, $s2                    # Pass base address of padded matrix
    move $a1, $s3                    # Pass base address of kernel matrix
    move $a2, $t3                    # Pass startRow
    move $a3, $t4                    # Pass startCol
    jal  dotProduct                  # Call dotProduct, result will be in $f0

    # Restore the saved registers
    lw $t0, 0($sp)                   # Restore $t0 (row index)
```

```
    lw $t1, 4($sp)    #restore $t1 (output size)
    lw $t2, 8($sp)                    # Restore $t2 (column index)
    lw $t3, 12($sp)                    # Restore $t3 (startRow)
    lw $t4, 16($sp)                   # Restore $t4 (startCol)
    addi $sp, $sp, 20                 # Deallocate stack space

    # Store the result in output matrix
    swc1 $f0, 0($s7)                  # Store the floating-point result
    addi $s7, $s7, 4                  # Move to next output matrix position
    addi $t2, $t2, 1                  # Increment j
    j    conv_col_loop

nextRow:
    addi $t0, $t0, 1                  # Increment i
    j    conv_row_loop
```

Dot product function:

```
# DotProduct Function
# Arguments:
#   $a0 - base address of padded matrix
#   $a1 - base address of kernel matrix
#   $a2 - startRow
#   $a3 - startColumn
# Returns:
#   $f0 - dot product result
dotProduct:
    # Initialize sum to 0.0
    li   $t5, 0                       # Integer 0
    mtc1 $t5, $f0                     # Move integer 0 to floating-point register $f0

    # Kernel index variables
    li   $t6, 0                       # i = 0 for kernel matrix row index
dot_product_row:
    lw   $t1, M      # Load size of kernel matrix (M)
    bge  $t6, $t1, dot_product_done    # If i >= kernel size, finish dot product

    li   $t7, 0                       # j = 0 for kernel matrix column index
dot_product_column:
    bge  $t7, $t1, next_dot_row       # If j >= kernel size,
```

```
go to next row

    # Calculate padded matrix element address
    lw   $t3, paddedSize      # Load size of padded matrix
    add  $t0, $t6, $a2               # Add startRow to the row index i (padded matrix row)
    add  $t2, $t7, $a3               # Add startCol to the column index j (padded matrix column)
    mul  $t4, $t0, $t3               # Row offset = (i + startRow) * sizeOfPaddedMatrix
    add  $t4, $t4, $t2               # Add column offset (j + startCol)
    sll  $t4, $t4, 2                 # Multiply by 4 (each element is 4 bytes)
    add  $t4, $t4, $a0               # Add base address of padded matrix
    lwc1 $f1, 0($t4)                 # Load floating-point element from padded matrix

    # Calculate kernel matrix element address
    mul  $t5, $t6, $t1               # Row offset = i * sizeOfKernelMatrix (for kernel matrix)
    add  $t5, $t5, $t7               # Add column offset (j) for kernel matrix
    sll  $t5, $t5, 2                 # Multiply by 4 (each element is 4 bytes)
    add  $t5, $t5, $a1               # Add base address of kernel matrix
    lwc1 $f2, 0($t5)                 # Load floating-point element from kernel matrix

    # Perform multiplication and accumulate the sum
    mul.s $f3, $f1, $f2              # Multiply padded matrix and kernel matrix elements
    add.s $f0, $f0, $f3              # Add to the sum

    addi $t7, $t7, 1                 # Increment kernel matrix column index (j)
    j    dot_product_column

next_dot_row:
    addi $t6, $t6, 1                 # Increment kernel matrix row index (i)
    j    dot_product_row

dot_product_done:
    jr   $ra                         # Return from function,
```

```
result in $f0
```

4.5 Process the output matrix to buffer of character function

```
write_to_file_function:
    # Load array information
    move $t0, $s4           # $t0 points to start of float_array
    lw $t1, outputSize              # Load the size of the array into $t1
    mul $t1, $t1, $t1 #outputSize * outputSize
    la $s0, bufferOutput            # Pointer to bufferOutput to store the final formatted result
    #la $s1, digitBufferForIntegerPart
    li $s7, 0                       # Initialize index to 0

loop_array:
    # Check if we reached the end of the array
    beq $s7, $t1, done              # If index >= array size, we're done

    # Load the next floating-point number
    l.s $f0, 0($t0)                 # Load float from array into $f0
    l.s $f4, num_zero                    # Load 0.0 into $f4
    c.lt.s $f0, $f4                  # Check if $f0 < 0
    bc1f positive                    # If not negative, branch to positive

negative:
    # Number is negative, so store '-' sign
    li $t4, '-'                      # ASCII for '-'
    sb $t4, 0($s0)                   # Store '-' in bufferOutput
    addiu $s0, $s0, 1                # Advance buffer pointer for the next character

    # Take absolute value
    neg.s $f0, $f0                   # Negate $f0 to make it positive

positive:
    #check the integer part here
    #if the first digit is zero: no need to extract
    cvt.w.s $f2, $f0                # Convert float in $f0 to integer
```

```
    mfc1 $t4, $f2                    # Move integer part to $t4

    # Check if the integer part is zero
    bne $t4, 0, start_extract_integer    # If integer part is not zero, skip to extraction

    mov.s $f12, $f0
    jal is_increase_integer_part

    beq $v0, 1, increase_int_part_zero
    j dont_increase_int_part_zero
increase_int_part_zero:
    li $t5, '1'                      # Load ASCII '0'
    sb $t5, 0($s0)                   # Store '0' in bufferOutput
    addiu $s0, $s0, 1
    j parsingDecimalPart
dont_increase_int_part_zero:
    #write 0 to buffer output
    li $t5, '0'                      # Load ASCII '0'
    sb $t5, 0($s0)                   # Store '0' in bufferOutput
    addiu $s0, $s0, 1                # Advance bufferOutput pointer
parsingDecimalPart:
    #passing number to parsing decimal part
    li $a0, '.'                      # ASCII for decimal point
    sb $a0, 0($s0)                   # Store decimal point in bufferOutput
    addiu $s0, $s0, 1                # Move bufferOutput pointer

    move $a0, $s0                    # Base address of bufferOutput
    mov.s $f12, $f0  # Load floating point value into $f12

    # Step 2: Extract and round decimal part
    jal extract_decimal_part         # Call function to extract and round the first decimal part
    #else passing the number to parsing
    move $s0, $v0
    j update_index
    # Step 1: Extract integer part and reverse order

start_extract_integer:

    mov.s $f12, $f0
    jal is_increase_integer_part

    beq $v0, 1, increase_int_part
```

```
    j dont_increase_int_part
increase_int_part:
    addi $t4, $t4,1

dont_increase_int_part:

    move $a0, $t4
    la   $a1, digitBufferForIntegerPart
    jal extract_integer_part      # Call function to extract integer part into bufferOutput

    # Reverse digitBuffer to get digits in correct order
    la $a0, digitBufferForIntegerPart           # Reset pointer to start of digitBuffer
    move $a1, $v0           # Number of digits in digitBuffer
    jal reverse_buffer      # Call reverse_buffer function

    move $a0, $s0           # Current pointer in bufferOutput
    la $a1, digitBufferForIntegerPart    # Base address of digitBuffer
    move $a2, $v0               # Number of digits to write
    jal writeDigitToBuffer # Call the function

    # The updated pointer of bufferOutput will be in $v0
    move $s0, $v0           # Update bufferOutput pointer in $s0 for further use
    addi $s0, $s0, 1     #space for the dot
    li $a0, '.'                 # ASCII for decimal point
    sb $a0, 0($s0)              # Store decimal point in bufferOutput
    addiu $s0, $s0, 1           # Move bufferOutput pointer

    move $a0, $s0               # Base address of bufferOutput
    mov.s $f12, $f0  # Load floating point value into $f12

    jal extract_decimal_part      # Call function to extract and round the first decimal part
    #update the pointer of buffer output
    move $s0, $v0

update_index:
    # Add a comma if it's not the last element in the array
    addiu $s7, $s7, 1           # Increment index
    bge $s7, $t1, skip_comma      # If this is the last element, skip adding comma

```

```
    li $t3, ','                    # ASCII for comma
    sb $t3, 0($s0)                 # Store comma in bufferOutput
    addiu $s0, $s0, 1              # Move buffer pointer forward by 1

skip_comma:
    # Move to the next float in the array
    addiu $t0, $t0, 4              # Move to the next float in the array
    j loop_array                   # Repeat for next array element

done:
    # Print bufferOutput by calling print_buffer function
    li $t3, '!'                    # ASCII for comma
    sb $t3, 0($s0)                 # Store comma in bufferOutput
    addiu $s0, $s0, 1              # Move buffer pointer forward by 1
    jal print_buffer

    jal write_to_file         # Call the function to write to file
```

4.5.1 is_increase_integer_part function

```
# Function: is_increase_integer_part
# Description: Extracts and rounds the first decimal digit from a floating-point number
#              passed in $f12 and stores it in bufferOutput.
# Parameters:
#   $f12 - floating-point number (passed as a parameter)
#   $a0 - base address of bufferOutput
# Return:
# $v1 - store the flag if 1 => increase integer

is_increase_integer_part:
    # Save registers to the stack
    addiu $sp, $sp, -20            # Allocate space on the stack
    sw $ra, 16($sp)                # Save return address
    sw $s0, 12($sp)                # Save bufferOutput base address
    sw $t9, 8($sp)                 # Save temporary register $t9
```

```
    sw $s6, 4($sp)                # Save temporary register
   $s6
    sw $s7, 0($sp)                # Save temporary register
   $s7

    li $v0, 0
    # Step 1: Extract the integer part
    cvt.w.s $f2, $f12             # Convert floating-point
   number in $f12 to integer
    mfc1 $t4, $f2                 # Move integer part to $t4

    # Step 2: Convert integer part back to float and subtract
     to get decimal part
    cvt.s.w $f2, $f2              # Convert integer part back
     to float
    l.s $f10, num_0.001
    sub.s $f4, $f12, $f2          # $f4 = decimal part of the
     original number
    add.s $f4, $f4, $f10
    # Step 3: Multiply decimal part by 100 to shift the first
     two decimal digits
    l.s $f6, num_100              # Load 100.0 into $f6

    mul.s $f4, $f4, $f6           # $f4 = decimal part * 100

    # Step 4: Convert to integer to get the first two decimal
     digits
    cvt.w.s $f8, $f4              # Convert $f4 to integer
    mfc1 $t9, $f8                 # Move first two decimal
   digits to $t9

    # Step 5: Extract first and second decimal digits
    div $t9, $t9, 10              # Divide by 10 to get first
     and second digits
    mfhi $s6                      # $s6 = second digit
    mflo $t9                      # $t9 = first digit

    # Step 6: Check second digit for rounding
    li $s7, 5                     # Load 5 for rounding
   comparison
    bge $s6, $s7, update_decimal_part         # If second
   digit >= 5, round up
    j is_increase

update_decimal_part:
    addi $t9, $t9, 1              # Round up the first digit

```

```
is_increase:
    addi $t9, $t9, 48                 # Convert to ASCII
    #if the char is ":" store "0" to buffer
    beq $t9, 58, update_flag
    j dont_update
update_flag:
    addi $v0, $v0, 1
dont_update:
    # Restore registers from the stack
    lw $s7, 0($sp)                    # Restore $s7
    lw $s6, 4($sp)                    # Restore $s6
    lw $t9, 8($sp)                    # Restore $t9
    lw $s0, 12($sp)                   # Restore bufferOutput base address
    lw $ra, 16($sp)                   # Restore return address
    addiu $sp, $sp, 20                # Deallocate stack space
    jr $ra                             # Return to caller
```

4.5.2 extract_integer_part function

```
# Function: extract_integer_part
# Description: Extracts the integer part from a number in $a0, stores the digits in reverse order
#               in the buffer at $a1, and returns the count of digits.
# Parameters:
#   $a0 - Integer to extract digits from
#   $a1 - Base address of the buffer to store digits
#
# Returns:
#   $v0 - Number of digits extracted

extract_integer_part:
    # Save registers to the stack
    addiu $sp, $sp, -16               # Allocate space on the stack
    sw $ra, 12($sp)                   # Save return address
    sw $t4, 8($sp)                    # Save temporary register $t4
    sw $t5, 4($sp)                    # Save temporary register $t5
    sw $t6, 0($sp)                    # Save temporary register $t6

    # Initialize variables
    li $t5, 10                        # Divisor for extracting digits (10)
```

```
    li $t6, 0                         # Digit count, initialized to 0
    move $t4, $a0                     # Copy the integer to $t4 for processing
    move $t7, $a1                     # Base address of digit buffer in $t7

extract_digits:
    blez $t4, done_integer_part       # If $t4 is 0, we are done with the integer part

    # Extract the last digit
    div $t4, $t5                      # Divide $t4 by 10
    mfhi $t8                          # $t8 = remainder (last digit)
    mflo $t4                          # Update $t4 with quotient

    # Store the ASCII of the digit in the buffer
    addi $t8, $t8, 48                 # Convert digit to ASCII
    sb $t8, 0($t7)                    # Store ASCII character in the buffer at $t7
    addiu $t7, $t7, 1                 # Move buffer pointer forward
    addiu $t6, $t6, 1                 # Increment digit count
    j extract_digits                  # Repeat for the next digit

done_integer_part:
    # Return the number of digits extracted in $v0
    move $v0, $t6                     # $v0 = digit count

    # Restore registers from the stack
    lw $t6, 0($sp)                    # Restore $t6
    lw $t5, 4($sp)                    # Restore $t5
    lw $t4, 8($sp)                    # Restore $t4
    lw $ra, 12($sp)                   # Restore return address
    addiu $sp, $sp, 16                # Deallocate stack space

    jr $ra                             # Return to caller
```

4.5.3 extract_decimal_part function

```
# Function: extract_decimal_part
# Description: Extracts and rounds the first decimal digit from a floating-point number
#              passed in $f12 and stores it in bufferOutput.
# Parameters:
#   $f12 - floating-point number (passed as a parameter)
```

```
#   $a0 - base address of bufferOutput

extract_decimal_part:
    # Save registers to the stack
    addiu $sp, $sp, -20          # Allocate space on the
   stack
    sw $ra, 16($sp)              # Save return address
    sw $s0, 12($sp)              # Save bufferOutput base
   address
    sw $t9, 8($sp)               # Save temporary register
   $t9
    sw $s6, 4($sp)               # Save temporary register
   $s6
    sw $s7, 0($sp)               # Save temporary register
   $s7

    # Set up $s0 as the base address of bufferOutput from $a0
    move $s0, $a0                # Move bufferOutput base
   address to $s0

    # Step 1: Extract the integer part
    cvt.w.s $f2, $f12            # Convert floating-point
   number in $f12 to integer
    mfc1 $t4, $f2                # Move integer part to $t4

    # Step 2: Convert integer part back to float and subtract
    to get decimal part
    cvt.s.w $f2, $f2             # Convert integer part back
    to float
    l.s $f10, num_0.001
    sub.s $f4, $f12, $f2         # $f4 = decimal part of the
    original number
    add.s $f4, $f4, $f10
    # Step 3: Multiply decimal part by 100 to shift the first
    two decimal digits
    l.s $f6, num_100             # Load 100.0 into $f6

    mul.s $f4, $f4, $f6          # $f4 = decimal part * 100

    # Step 4: Convert to integer to get the first two decimal
    digits
    cvt.w.s $f8, $f4             # Convert $f4 to integer
    mfc1 $t9, $f8                # Move first two decimal
   digits to $t9

    # Step 5: Extract first and second decimal digits
    div $t9, $t9, 10             # Divide by 10 to get first
```

```
    and second digits
    mfhi $s6                        # $s6 = second digit
    mflo $t9                        # $t9 = first digit

    # Step 6: Check second digit for rounding
    li $s7, 5                       # Load 5 for rounding comparison
    bge $s6, $s7, round_up          # If second digit >= 5, round up
    j store_first_decimal           # Otherwise, store the first decimal as is

round_up:
    addi $t9, $t9, 1                # Round up the first digit

store_first_decimal:
    addi $t9, $t9, 48               # Convert to ASCII
    #if the char is ":" store "0" to buffer
    beq $t9, 58, store_zero
    j store_to_buffer
store_zero:
    li $t9, 48
    addi $v1, $v1, 1
    j store_to_buffer
store_to_buffer:
    sb $t9, 0($s0)                  # Store in bufferOutput at $s0
    addiu $s0, $s0, 1               # Move bufferOutput pointer
    move $v0, $s0

    # Restore registers from the stack
    lw $s7, 0($sp)                  # Restore $s7
    lw $s6, 4($sp)                  # Restore $s6
    lw $t9, 8($sp)                  # Restore $t9
    lw $s0, 12($sp)                 # Restore bufferOutput base address
    lw $ra, 16($sp)                 # Restore return address
    addiu $sp, $sp, 20              # Deallocate stack space

    jr $ra                           # Return to caller
```

4.5.4 reverse_buffer function

```
# Function: reverse_buffer
# Description: Reverses the contents of a buffer to store digits in the correct order.
# Parameters:
```

```
#   $a0 - base address of the buffer
#   $a1 - number of digits in the buffer

reverse_buffer:
    # Save registers to the stack
    addiu $sp, $sp, -16             # Allocate space on stack
    sw $ra, 12($sp)                 # Save return address
    sw $t5, 8($sp)                  # Save temporary register $t5
    sw $t6, 4($sp)                  # Save temporary register $t6
    sw $t7, 0($sp)                  # Save temporary register $t7

    # Set up pointers
    move $t5, $a0                   # Start pointer (base address of the buffer)
    add $t6, $t5, $a1               # End pointer (base address + number of digits)
    subi $t6, $t6, 1                # Adjust to point to the last valid digit

reverse_loop:
    bge $t5, $t6, end_reverse       # If pointers meet or cross, we're done

    # Swap the values at $t5 and $t6
    lb $t7, 0($t5)                  # Load value at the start
    lb $t8, 0($t6)                  # Load value at the end
    sb $t8, 0($t5)                  # Store end value at start
    sb $t7, 0($t6)                  # Store start value at end

    # Move pointers inward
    addiu $t5, $t5, 1               # Move start pointer forward
    subi $t6, $t6, 1                # Move end pointer backward
    j reverse_loop                  # Repeat the loop

end_reverse:
    # Restore registers from the stack
    lw $t7, 0($sp)                  # Restore $t7
    lw $t6, 4($sp)                  # Restore $t6
    lw $t5, 8($sp)                  # Restore $t5
    lw $ra, 12($sp)                 # Restore return address
    addiu $sp, $sp, 16              # Deallocate stack space

    jr $ra                           # Return to caller
```

4.5.5 write_digit_to_buffer function

```
# Function: writeDigitToBuffer
# Description: Copies digits from digitBuffer to bufferOutput
    .
# Parameters:
#   $a0 - Current pointer of bufferOutput
#   $a1 - Base address of digitBuffer
#   $a2 - Number of digits to write

writeDigitToBuffer:
    # Save registers to the stack
    addiu $sp, $sp, -12             # Allocate space on the
   stack
    sw $ra, 8($sp)                  # Save return address
    sw $t0, 4($sp)                  # Save temporary register
   $t0
    sw $t1, 0($sp)                  # Save temporary register
   $t1

    # Initialize loop variables
    move $t0, $a1                   # $t0 points to base
   address of digitBuffer
    move $t1, $a0                   # $t1 points to current
   pointer of bufferOutput
    move $t2, $a2                   # Number of digits to copy

copyLoop:
    # Check if all digits are copied
    blez $t2, end_write             # If $t2 <= 0, end the
   function

    # Load a byte from digitBuffer and store it in
   bufferOutput
    lb $t3, 0($t0)                  # Load digit from
   digitBuffer
    sb $t3, 0($t1)                  # Store digit in
   bufferOutput

    # Advance pointers
    addiu $t0, $t0, 1               # Move to the next digit in
    digitBuffer
    addiu $t1, $t1, 1               # Move to the next position
    in bufferOutput
    subi $t2, $t2, 1                # Decrease the count of
   digits to copy
    j copyLoop                     # Repeat the loop
```

```

end_write:
    # Update bufferOutput pointer (returning the updated pointer in $v0)
    subi $t1, $t1, 1
    move $v0, $t1                    # $v0 now points to the new position in bufferOutput

    # Restore registers from the stack
    lw $t1, 0($sp)                   # Restore $t1
    lw $t0, 4($sp)                   # Restore $t0
    lw $ra, 8($sp)                   # Restore return address
    addiu $sp, $sp, 12               # Deallocate stack space

    jr $ra
```

4.6 write_to_file function

```
# Function: write_to_file
# Description: Opens a file and writes each character from bufferOutput to the file.
#              Stops when '!' is encountered. Replaces ',' with a newline.
# Parameters:
#   $a0 - Base address of bufferOutput

write_to_file:
    # Save registers to the stack
    addiu $sp, $sp, -12              # Allocate space on stack
    sw $ra, 8($sp)                   # Save return address
    sw $t0, 4($sp)                   # Save temporary register $t0
    sw $t1, 0($sp)                   # Save temporary register $t1

    # Step 1: Open the file in write mode
    li $v0, 13                       # Syscall for opening a file
    la $a0, fout                 # File name
    li $a1, 1                        # Flag: 1 for write-only
    li $a2, 0                        # Mode: 0 (not applicable for writing)
    syscall
    move $t0, $v0                    # Store file descriptor in $t0

```

```
    # Check if file opened successfully
    #bltz $t0, file_error          # If file descriptor is
   negative, jump to error

    # Step 2: Loop through bufferOutput
    la $t1, bufferOutput                 # $t1 points to the
   current character in bufferOutput

write_loop:
    lb $t2, 0($t1)                  # Load the current
   character from bufferOutput

    # Check for end condition '!'
    li $t3, '!'                     # Load '!' to compare
    beq $t2, $t3, end_write_to_output       # If character is
    '!', end writing

    # Check if character is a comma ','
    li $t3, ','                     # Load ',' to compare
    beq $t2, $t3, write_space    # If character is ',', write
   a newline

    # Otherwise, write the character to file
    li $v0, 15                      # Syscall for writing to
   file
    move $a0, $t0                   # File descriptor
    move $a1, $t1                   # Address of the character
   to write
    li $a2, 1                       # Number of bytes to write
    syscall

    j next_char                     # Go to next character

write_space:
    # Write newline character instead of comma
    li $v0, 15                      # Syscall for writing to
   file
    move $a0, $t0                   # File descriptor
    la $a1, space                  # Address of the newline
   character
    li $a2, 1                       # Number of bytes to write
    syscall

next_char:
    addiu $t1, $t1, 1               # Move to the next
   character in bufferOutput
    j write_loop                    # Repeat loop
```

```

end_write_to_output:
    # Step 3: Close the file
    li $v0, 16                        # Syscall for closing file
    move $a0, $t0                     # File descriptor
    syscall

    # Restore registers from the stack
    lw $t1, 0($sp)                    # Restore $t1
    lw $t0, 4($sp)                    # Restore $t0
    lw $ra, 8($sp)                    # Restore return address
    addiu $sp, $sp, 12                # Deallocate stack space

    jr $ra                             # Return to caller
```

5 Result

5.1 Testcase 1

Input of testcase 1:

```
5.0 3.0 1.0 2.0
-9.1 8.5 1.5 -5.9 5.0 0.8 3.1 4.6 -9.1 -2.7 -9.6 -1.8 -3.6
   -8.9 -9.7 -7.8 -0.2 7.0 3.1 -4.8 -0.8 9.1 -1.5 -6.2 6.3
-9.6 -8.5 -2.1 -7.2 9.8 0.6 -5.3 6.2 -5.2
```

The result in mars mips simulation:

```
The image matrix is:
-9.1 8.5 1.5 -5.9 5.0
0.8 3.1 4.6 -9.1 -2.7
-9.6 -1.8 -3.6 -8.9 -9.7
-7.8 -0.2 7.0 3.1 -4.8
-0.8 9.1 -1.5 -6.2 6.3

The image matrix after padded is:
0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 -9.1 8.5 1.5 -5.9 5.0 0.0
0.0 0.8 3.1 4.6 -9.1 -2.7 0.0
0.0 -9.6 -1.8 -3.6 -8.9 -9.7 0.0
0.0 -7.8 -0.2 7.0 3.1 -4.8 0.0
0.0 -0.8 9.1 -1.5 -6.2 6.3 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0
The Kernel Matrix
-9.6 -8.5 -2.1
-7.2 9.8 0.6
-5.3 6.2 -5.2
```

Figure 10: *The image, padded and kernel matrix of input file*

```
Start to perform convolutional operation...
The output matrix is:
-95.240005 9.369995 122.96999
-155.79001 -49.070007 33.140015
64.340004 -148.03001 117.42
The output matrix is:
-95.2,9.4,123.0,-155.8,-49.1,33.1,64.3,-148.0,117.4!

-- program is finished running --
```

Figure 11: *Result of performing convolution operation and buffer output*

Output in output.txt file:

```
output.txt - Notepad
File Edit Format View Help
-95.2 9.4 123.0 -155.8 -49.1 33.1 64.3 -148.0 117.4
```

Figure 12: *output.txt file*

5.2 Testcase 2

Input of testcase 2:

```
5.0 2.0 3.0 1.0
-2.0 -6.6 -4.6 -6.4 8.4 -2.6 4.9 6.6 0.9 7.9 6.4 -3.5 -7.7
    -8.3 -2.0 6.8 -1.0 -7.8 -6.2 -4.5 -8.3 -1.2 -2.6 -1.0 -0.8
2.6 9.7 7.6 -6.6
```

The result in mars mips simulation:

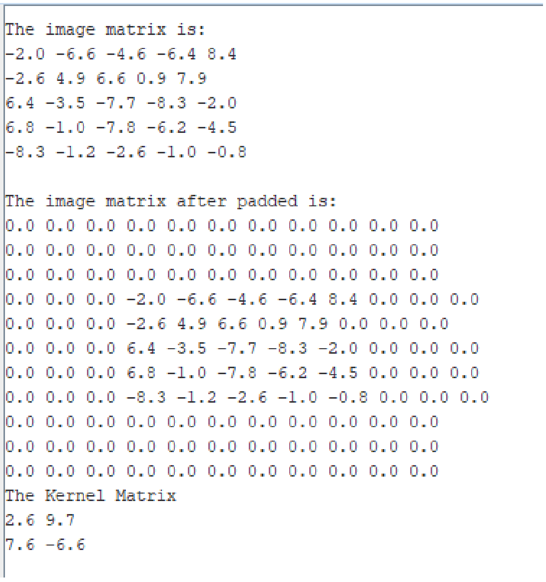

Figure 13: *The image, padded and kernel matrix of input file*

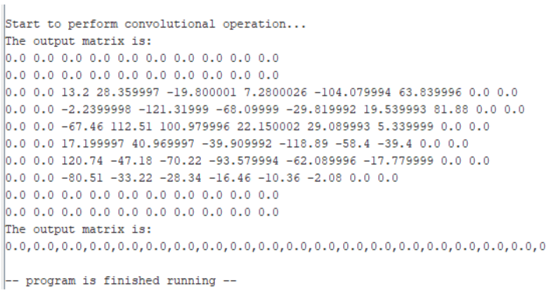

Figure 14: *Result of performing convolution operation and buffer output*

Output in output.txt file:

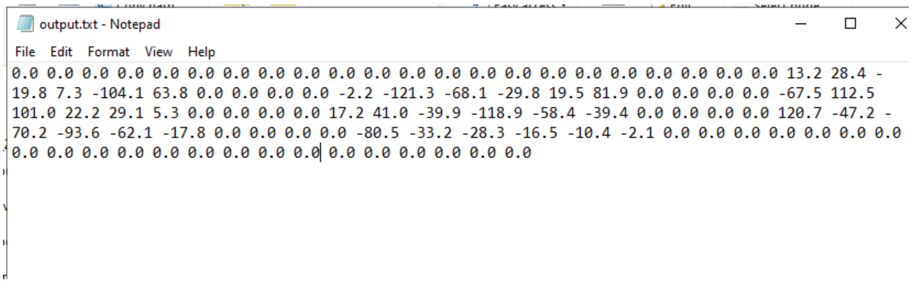

Figure 15: *output.txt file*

5.3 Testcase 3

Input of testcase 3:

```
7.0 4.0 1.0 2.0
-4.4 -1.3 8.7 7.1 3.6 6.3 5.3 8.8 -4.3 1.5 6.6 -2.7 3.5 9.4 5.9 -5.8 3.4 8.6 -4.1 -0.0 1.5 0.9 0.7 -2.4 -2.3 -8.7 -9.1 -5.7 -0.5 2.4 -7.5 1.0 -0.6 3.1 -5.8 9.9 -0.9 5.3 -2.2 -1.2 -7.6 4.3 1.4 -5.2 -7.6 4.0 9.0 8.7 1.3
-7.7 6.9 -4.2 9.1 6.8 0.8 1.4 8.6 1.3 2.0 -0.1 -7.7 0.4 -6.0 -0.5 1.0
```

The result in mars mips simulation:

```
-4.4 -1.3 8.7 7.1 3.6 6.3 5.3
8.8 -4.3 1.5 6.6 -2.7 3.5 9.4
5.9 -5.8 3.4 8.6 -4.1 -0.0 1.5
0.9 0.7 -2.4 -2.3 -8.7 -9.1 -5.7
-0.5 2.4 -7.5 1.0 -0.6 3.1 -5.8
9.9 -0.9 5.3 -2.2 -1.2 -7.6 4.3
1.4 -5.2 -7.6 4.0 9.0 8.7 1.3

The image matrix after padded is:
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 -4.4 -1.3 8.7 7.1 3.6 6.3 5.3 0.0
0.0 8.8 -4.3 1.5 6.6 -2.7 3.5 9.4 0.0
0.0 5.9 -5.8 3.4 8.6 -4.1 -0.0 1.5 0.0
0.0 0.9 0.7 -2.4 -2.3 -8.7 -9.1 -5.7 0.0
0.0 -0.5 2.4 -7.5 1.0 -0.6 3.1 -5.8 0.0
0.0 9.9 -0.9 5.3 -2.2 -1.2 -7.6 4.3 0.0
0.0 1.4 -5.2 -7.6 4.0 9.0 8.7 1.3 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
The Kernel Matrix
-7.7 6.9 -4.2 9.1
6.8 0.8 1.4 8.6
1.3 2.0 -0.1 -7.7
0.4 -6.0 -0.5 1.0
```

Figure 16: *The image, padded and kernel matrix of input file*

```
Start to perform convolutional operation...
The output matrix is:
46.859993 25.439997 65.55
132.78001 39.42 90.55
-114.43 -15.4900055 -188.25
The output matrix is:
46.9,25.4,65.6,132.8,39.4,90.6,-114.4,-15.5,-188.3!

-- program is finished running --
```

Figure 17: *Result of performing convolution operation and buffer output*

Output in output.txt file:

```
output.txt - Notepad
File Edit Format View Help
46.9 25.4 65.6 132.8 39.4 90.6 -114.4 -15.5 -188.3
```

Figure 18: *output.txt file*

5.4 Testcase 4

Input of testcase 4:

```
4.0 2.0 3.0 1.0
3.5 -8.9 -4.3 9.8 -5.1 -8.3 8.5 4.2 6.5 8.5 -9.3 9.6 -0.9
    -6.7 4.1 6.5
4.2 8.8 -8.2 7.8
```

The result in mars mips simulation:

```
The image matrix is:
3.5 -8.9 -4.3 9.8
-5.1 -8.3 8.5 4.2
6.5 8.5 -9.3 9.6
-0.9 -6.7 4.1 6.5

The image matrix after padded is:
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 3.5 -8.9 -4.3 9.8 0.0 0.0 0.0
0.0 0.0 0.0 -5.1 -8.3 8.5 4.2 0.0 0.0 0.0
0.0 0.0 0.0 6.5 8.5 -9.3 9.6 0.0 0.0 0.0
0.0 0.0 0.0 -0.9 -6.7 4.1 6.5 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
The Kernel Matrix
4.2 8.8
-8.2 7.8
```

Figure 19: *The image, padded and kernel matrix of input file*

```
Start to perform convolutional operation...
The output matrix is:
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 27.300001 -98.119995 39.439995 111.700005 -80.36 0.0 0.0
0.0 0.0 -8.979998 -86.54001 59.14 31.24001 6.720001 0.0 0.0
0.0 0.0 5.8199997 -81.45999 -102.299995 223.8 -61.08 0.0 0.0
0.0 0.0 50.18 57.220005 40.77999 62.500008 -12.98 0.0 0.0
0.0 0.0 -7.92 -62.739998 7.9400043 74.42 27.3 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
The output matrix is:
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,27.3,

-- program is finished running --
```

Figure 20: *Result of performing convolution operation and buffer output*

Output in output.txt file:

```
output.txt - Notepad
File Edit Format View Help
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 27.3 -98.1 39.4 111.7
-80.4 0.0 0.0 0.0 0.0 -9.0 -86.5 59.1 31.2 6.7 0.0 0.0 0.0 0.0 5.8 -81.5 -102.3 223.8 -61.1 0.0 0.0
0.0 0.0 50.2 57.2 40.8 62.5 -13.0 0.0 0.0 0.0 0.0 -7.9 -62.7 7.9 74.4 27.3 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

Figure 21: *output.txt file*

5.5 Testcase 5

Input of testcase 5:

```
3.0 3.0 4.0 3.0
4.4 -7.2 -9.5 -1.6 -6.2 9.5 -4.3 -2.7 4.6
-9.5 5.3 5.9 -5.3 -3.4 1.9 -6.8 7.4 4.9
```

The result in mars mips simulation:

```
The image matrix is:
4.4 -7.2 -9.5
-1.6 -6.2 9.5
-4.3 -2.7 4.6

The image matrix after padded is:
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 4.4 -7.2 -9.5 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 -1.6 -6.2 9.5 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 -4.3 -2.7 4.6 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
The Kernel Matrix
-9.5 5.3 5.9
-5.3 -3.4 1.9
-6.8 7.4 4.9
```

Figure 22: *The image, padded and kernel matrix of input file*

```
Start to perform convolutional operation...
The output matrix is:
0.0 0.0 0.0
0.0 -70.86 -14.249996
0.0 -38.72 -43.7
The output matrix is:
0.0,0.0,0.0,0.0,-70.9,-14.3,0.0,-38.7,-43.7!

-- program is finished running --
```

Figure 23: *Result of performing convolution operation and buffer output*

Output in output.txt file:

```
output.txt - Notepad
File Edit Format View Help
0.0 0.0 0.0 0.0 -70.9 -14.3 0.0 -38.7 -43.7
```

Figure 24: *output.txt file*

5.6 Testcase with input error

Testcase 1

```
3.0 4.0 4.0 1.0
-4.7 9.5 9.1 -8.1 -2.4 -6.2 -9.8 -3.8 6.9 2.6 2.7
-4.1 -7.4 5.9 2.4 2.7 6.2 2.5 4.0
```

The result in mars mips simulation:

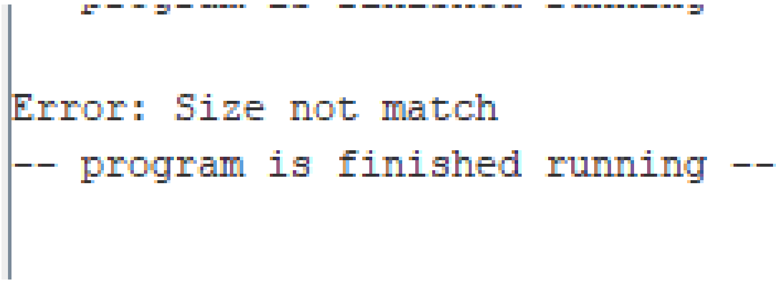

Figure 25: *The mars mips simulation result*

Testcase 2

```
2.0 5.0 5.0 0.0
-4.6 9.9 -9.9 9.7 1.1 -3.1 9.9 -2.9 -7.5
-0.1 -2.9 7.8 -4.0
```

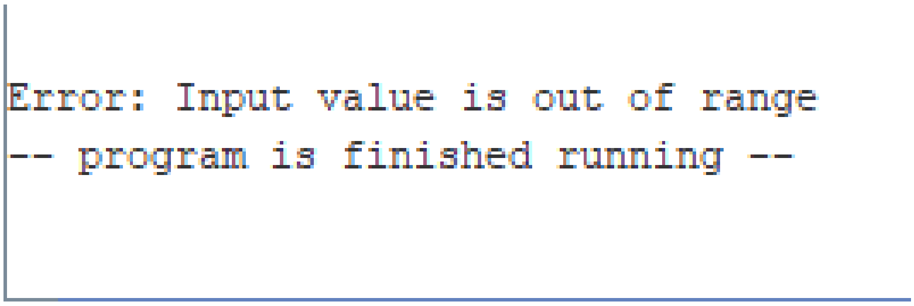

Figure 26: *The mars mips simulation result*