

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



COMPUTER ARCHITECTURE

Major assignment report

CONVOLUTION OPERATION

Advisor(s): Nguyen Thien An

Student(s): Pham Tran Gia Phu 2352921

HO CHI MINH CITY, NOVEMBER 2024



Contents

1	Abstract	5
2	Introduction	5
3	Methodology	5
3.1	Construct matrices from read buffer	6
3.1.1	String to float conversion	8
3.1.2	Loading configuration	10
3.1.3	Add padding to image	10
3.2	Convolve constructed matrices	11
3.3	Construct write buffer from the result matrix	12
4	Images of test runs	14

List of Figures

3.1	General steps performed by the program	6
3.2	Matrix building algorithm	7
4.1	Sample test on LMS	14
4.2	Output of sample test on LMS	14
4.3	Standard testcase	15
4.4	Output	15
4.5	Another Standard testcase	16
4.6	Output	16
4.7	Test where size of image is smaller than kernel's	17
4.8	Output of test where size of image is smaller than kernel's	17

List of Tables

3.1	Conversion of whole number part	9
3.2	Conversion of decimal part	9



Listings

3.1 Pseudocode for convolution operation	11
--	----



1 Abstract

This report covers the implementation and analysis of convolution operations using MIPS Assembly. In this assignment, usage of the following pieces of software and tools were involved:

- Visual Studio Code
- MARScore extension for Visual Studio Code
- MARS 4.5

2 Introduction

Convolutional neural network (CNN) is one of the most widely used type of network used in deep learning. Its usage can be seen in a wide range of applications, particularly in fields that involve image and video analysis. However advanced a CNN model maybe, it always stems from matrix computations and most notably, convolution operations.

The convolution operation involves sliding a filter, also known as a kernel, across the input image. This filter is a small matrix of weights that is applied to a local region of the input matrix, producing a single value in the output feature map. The process is repeated across the entire matrix, allowing the network to learn various features of the matrix. In the case of image analysis, these features can include edges, textures, and patterns. The mathematical operation performed is essentially a dot product between the filter and the receptive field of the input image.

The process involves taking a small matrix, called a kernel or filter, and sliding it over an input matrix, such as an image. At each position, the dot product of the kernel and the overlapping region of the input matrix is computed, and the result is stored in the output matrix. This process is repeated for all positions of the kernel over the input matrix, effectively blending the kernel with the input to highlight specific features like edges or textures.

3 Methodology

In this section, we discuss the core methods used, including reading and processing test files, convolve matrices and writing calculated results to output files with the use

of MIPS Assembly. First, we should take a look at the general steps performed by the program and then take a deeper analysis on each individual step.

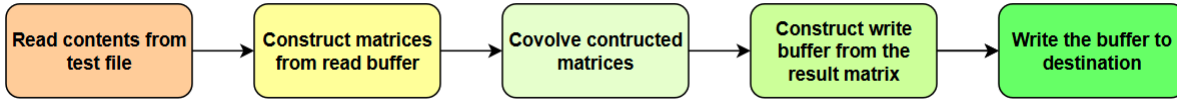


Figure 3.1: General steps performed by the program

3.1 Construct matrices from read buffer

Reading contents of a file was an easy task, but the next process poses far greater challenges. Base on the requirements of this assignment, we first need to construct three matrices from the data acquired from file reading step. These matrices consist of:

- Configuration matrix (**config**): This matrix contains values for the four variables N, M, P and S, which is then used to then load each element into separate variables.
- Image matrix (**image**): This matrix contains values that the image should have.
- Kernel matrix (**kernel**): This matrix contains values that the kernel should have.

Intentionally putting N, M, P and S inside the same matrix allows us to reuse the same procedure **build_matrix**, which was initially used solely to build **image** and **kernel**.

The algorithm we used to construct these matrices from **input_buffer** requires us to specify two pointers, one to the matrix we want to construct and another one to where reading starts in the buffer, and how many elements is needed. For **config**, the read pointer will point to the start of the file, to the start of the second row if constructing **image** and to the start of the third row if the matrix is **kernel**. Below is a flowchart capturing core steps done by the algorithm. Note that **character** refers to individual letter or digit or symbol in the string while **element** means whole word or number. For example, let us assume the string "12.34 5.6", for this particular case, '1', '2', '3', '4', '5', '6', ' ' and '.' are characters while "12.34" and "5.6" are elements.

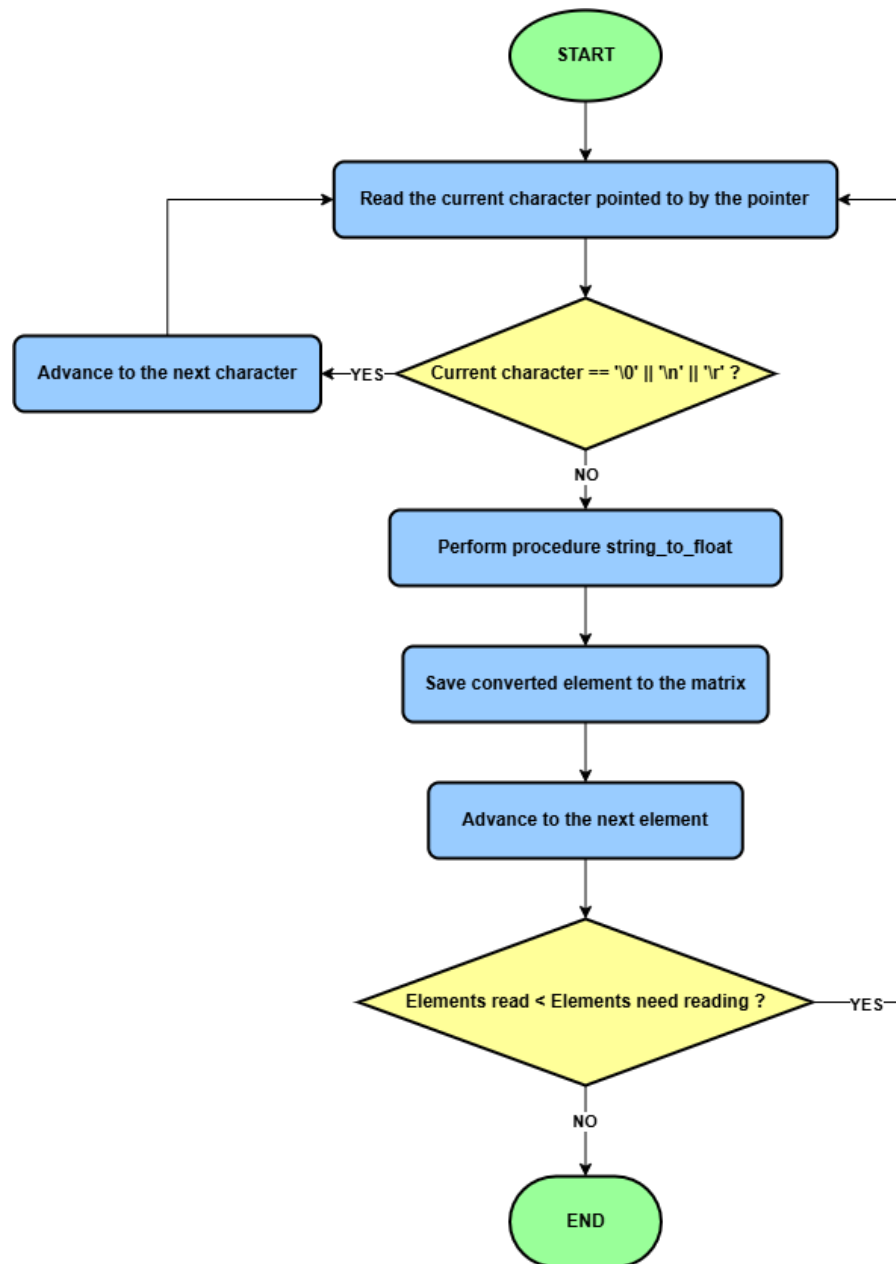


Figure 3.2: Matrix building algorithm

To avoid confusion, **Advance to the next character** means the read pointer only have to move forward one byte to read the next character in the string, **Advance to the next element** not only does the same task but also move the matrix pointer four bytes.

3.1.1 String to float conversion

During the **Perform procedure string_to_float** involves a separate algorithm converting an element in string format to a suitable floating point number. Let us now analyze the mathematical properties of such a number before building the algorithm. A floating point number such as 123.456 can be presented by sum of products between individual digit and a powers of ten.

$$\begin{aligned} 123.456 &= 100 + 20 + 3 + 0.4 + 0.05 + 0.006 \\ &= 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2} + 6 \cdot 10^{-3} \end{aligned}$$

For negative numbers, we can just set up a flag if our read pointer encounter a negative sign. The **string_to_float** algorithm only need to convert its absolute value and negate the result before returning if and only if the negative flag is raised. Having that out of the way, let us shift our focus on the actual string-float conversion. We initially assign a register the value 0.0, acting as our returning value, another register with the value 10.0 as a way to compute $10^n (n \geq 0)$ through repeated multiplications and one last register the value 10.0 to compute $10^{-m} (m > 0)$ via divisions. Since whole number and decimal part involve different calculation, the procedure will have to be split into two halves, each handle one of said parts. The stop conditions for both would be if the current character is either '`\0`' or '`\n`' or '`\r`' or ' ' and the transition condition from converting whole number part to decimal part would be if the current character is '.'.

While converting the whole numbers, we must first convert each digit from ASCII code to its corresponding numerical value by subtracting 48. But our matrices need the values to be in float, and the initial conversion produces an integer. To solve this problem, we put said integer into the stack and uses `lwc1` to transfer it from the stack to a f-register, then use `cvt.s.w` to convert the integer into how it would presented in IEEE-754 format. We advance the read pointer and repeat this process until it reaches one of the stop condition or decimal symbol, return result for the first case and jump to converting decimal part for the latter one. For each converted digit, shift the result to the left by multiply it by 10 and put new digit to the least significant digit by addition. This process is similar to how we multiply each digit by $10^n (n \geq 0)$ and n gets smaller as the iteration goes on. Below is a table illustrating the value of result register over each iteration.



Table 3.1: Conversion of whole number part

Values of result over each iteration		
Iteration	Current character	Result
0	1	0.0
1	1	$0.0 \cdot 10 + 1 = 1.0$
2	2	$1.0 \cdot 10 + 2 = 12.0$
3	3	$12.0 \cdot 10 + 3 = 123.0$
4	.	123.0

Converting decimal part take a similar process though differ in the calculation of result. Instead of multiplication by 10, converting the decimals do division to simulates the action of multiply each digit with $10^{-m} (m > 0)$ and as stated above, m gets bigger as the iteration go on so we multiply the temporary 10-register by 10, which is equivalent to $m = m + 1$. We put the new digit into our result in a similar manner as we do with whole numbers.

Table 3.2: Conversion of decimal part

Values of result over each iteration		
Iteration	Current character	Result
0	4	123.0
1	4	$123.0 + 4/10 = 123.4$
2	5	$123.4 + 5/100 = 123.45$
3	6	$123.45 + 6/1000 = 123.456$
4	\0	123.456

Since the procedure reached one of the stop condition, it checks for negative flag before return to caller. If the flag is raised, returns the invert of result, else returns result itself. The whole process of reading character from string and converting elements to corresponding floating point numbers repeats until it reaches the number of elements user desired.

3.1.2 Loading configuration

Having mentioned prior, the values of N, M, P and S are stored in the matrix **config**, thus a procedure is needed to transfer each of them to their corresponding variable and use those to calculate the size of output matrix. The first task can be done by moving the matrix pointer 4 bytes every time we finish loading one configuration and convert each from IEEE-754 format to their corresponding integer form using **cvt.w.s**. Moving on to the second task, the size of output is calculated by this formula:

$$\lfloor \frac{N' - M}{S} \rfloor + 1 \quad (3.1)$$

$$N' = 2P + N \quad (3.2)$$

Flooring the $\frac{N'-M}{S}$ is essentially taking quotient of the division using **mflo**. However, there are situations where the size of output matrix is less than or equals to 0, such cases are invalid and can be ruled out by investigate $N' - M$, if the subtraction results in a negative number, we write to output "Error: size not match" and terminate, otherwise continue the calculation and return to caller.

3.1.3 Add padding to image

For task, it best to explain the algorithm via an example. Assume two of the configurations are $N = 3$, and $P = 1$ and the image matrix is

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

and due to how we initialized the matrices in **.data**, all of them will be in largest possible size with 0 as their initial elements, the padded image first appears like so:

$$I_i^P = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Base on the assumption, the padded image should be like the one below after calling `build_padded_image`.

$$I_f^P = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The algorithm used by the procedure first calculate the location where the first element of I would be if it was put inside I^P using formula (3.3), then begin copying remaining elements in I to I^P . For elements in the same row, we only need to move the matrix pointer 4 bytes every time, but to advance to the next row, we have to move it by $2P \cdot 4$ bytes, skipping the padding layers on the side.

$$D = (N + 2P) \cdot P + P \quad (3.3)$$

Explanation for the above formula, $N + 2P$ is the size of a row in any padded image, multiply it with P is equivalence to skipping all the padding layers on the top of the matrix and adding P is the same as moving through those layers on the side.

3.2 Convolve constructed matrices

The algorithm performing convolution operation is based on this piece of C++ code.

Listing 3.1: Pseudocode for convolution operation

```
1 for every row i in output matrix {
2   for every column j in output matrix {
3     sum = 0
4     for every row k in kernel {
5       for every column l in kernel {
6         row = i * stride + k
7         col = j * stride + l
8         sum += padded_mat[row][col] * kernel[k][l]
9       }
10    }
11    out[i][j] = sum
12  }
13 }
```

For each position (i, j) in the output matrix:

- Calculate the region of the input matrix covered by the kernel.
- Multiply each element of the kernel with the corresponding element of the input matrix.
- Sum up these multiplications to get the value for the output matrix at (i, j) .

All the steps in the sample code are relatively easy to do in MIPS, except for getting a specific value at (i, j) in a matrix. In MIPS Assembly, information of matrices or 2D arrays are stored into a stream of numbers, one row after another. Getting an element at row i , we thus have to multiply it by 4 and the row length of the matrix we want to access then adding up the result with the matrix's address. Retrieving an element at column j is more simple, given that the row is specified and previously mentioned calculation done, we only need to add the result with j multiplied by 4 to access the right column. In short, getting any element's address specify by **matrix[i][j]** can be calculated using the following formulas:

$$i' = i \cdot \text{row_length} \cdot 4 \quad (3.4)$$

$$j' = i' + 4 \cdot j \quad (3.5)$$

$$\text{element_address} = \text{matrix_address} + i' + j' \quad (3.6)$$

3.3 Construct write buffer from the result matrix

Before writing to the output file, we have to convert our **out** matrix into a string with each element rounded to the one of decimal. The algorithm for this task loops over the matrix to get the elements and carry out following steps on each of them:

- If it is negative, write '-' to the buffer, else skip this step.
- Calculate its absolute value for float-string conversion.
- Extract the whole number part using **trunc.w.s** and move it to a temporary register.
- Call procedure **int_to_string** and write the converted number to buffer.
- Write to buffer '.' to create separation between whole number and decimal part in the string.



- Extract the decimal part, subtracting the original element by its previously calculated whole number part would do the trick.
- Since computers have precision issues when it comes to dealing with floating-point numbers and observation through various tests, we have to add an epsilon to the subtraction result in the last step to compensate.
- We only need one number as decimal so multiply the current isolated decimal part by ten and use **trunc.w.s** will give us the first decimal as an integer.
- Call **int_to_string** to convert that one decimal to string and write it to buffer.
- Write to buffer ' ' to separate elements in the matrix

One key factor in this algorithm is converting the whole number and decimal part into string, it is easier to convert each separately like how we extract individual one before calling the procedure **int_to_string**, this change the task into converting integer to string. Said conversion is straight forward, we divide the integer by ten and write the remainder into the buffer then continue as long as the quotient is non-zero. This approach is partly correct as the integer would be written backward, we can solve this by creating a temporary pointer and move it some distance to the right of the buffer one. As we write the remainder to where the temporary pointer is pointing at, shift it backward by one byte and keep track of how many digits were converted, continue this cycle until the quotient is zero. Now that our number is written correctly, we still need to move it to the location of the buffer pointer, this is done by transferring the digit at temporary pointer to buffer pointer then shift both forward by one byte and repeat the process until we have moved enough digits. Note that we always have to keep track of the number of digits and characters written to the buffer as it is crucial when we write the buffer to a file.

4 Images of test runs

A handful of testcases and their corresponding output were chosen to be present in this section.

```
===== Input =====
5.0 3.0 0.0 1.0
1.2 1.5 2.1 0.0 0.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0 1.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0
1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0

===== Image =====
1.2    1.5    2.1    0.0    0.0
0.0    1.0    1.0    1.0    0.0
0.0    0.0    1.0    1.0    1.0
0.0    0.0    1.0    1.0    0.0
0.0    1.0    1.0    0.0    0.0

===== Kernel =====
1.0    0.0    1.0
0.0    1.0    0.0
1.0    0.0    1.0

===== Added padding =====
1.2    1.5    2.1    0.0    0.0
0.0    1.0    1.0    1.0    0.0
0.0    0.0    1.0    1.0    1.0
0.0    0.0    1.0    1.0    0.0
0.0    1.0    1.0    0.0    0.0

===== Result =====
5.3    3.5    5.1
2.0    4.0    3.0
2.0    3.0    4.0
```

Figure 4.1: Sample test on LMS


```
Tests > Test_9 >  output_matrix.txt
1    5.3 3.5 5.1 2.0 4.0 3.0 2.0 3.0 4.0
```

Figure 4.2: Output of sample test on LMS

```

===== Input =====

4 3 3 3
1 1.2 -1.3 4.5 -5.0 3 3.5 6 -8.9 12 23.2 12 13 -14 -15 16.0
-3.0 -4 4.5 6 7.8 12 5 -0.5 12.0

===== Image =====

1.0    1.2    -1.3    4.5
-5.0    3.0    3.5    6.0
-8.9    12.0   23.2    12.0
13.0   -14.0  -15.0    16.0

===== Kernel =====

-3.0   -4.0    4.5
6.0    7.8    12.0
5.0    -0.5    12.0

===== Added padding =====

0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0
0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0
0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0
0.0    0.0    0.0    1.0    1.2   -1.3    4.5    0.0    0.0    0.0
0.0    0.0    0.0   -5.0    3.0    3.5    6.0    0.0    0.0    0.0
0.0    0.0    0.0   -8.9   12.0   23.2   12.0    0.0    0.0    0.0
0.0    0.0    0.0   13.0  -14.0  -15.0   16.0    0.0    0.0    0.0
0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0
0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0
0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0

===== Result =====

0.0    0.0    0.0
0.0   249.65002    82.5
0.0   -50.5   -48.0

```

Figure 4.3: Standard testcase

```

Tests > Test_6 > output_matrix.txt
1  0.0 0.0 0.0 0.0 249.7 82.5 0.0 -50.5 -48.0

```

Figure 4.4: Output

```
===== Input =====  
  
3 4 1 2  
-3.0 -4 4.5 6 7.8 12 5 -0.5 12.0  
  
1 1.2 -1.3 4.5 -5.0 3 3.5 6 -8.9 12 23.2 12 13 -14 -15 16.0  
  
===== Image =====  
  
-3.0    -4.0    4.5  
6.0     7.8     12.0  
5.0     -0.5    12.0  
  
===== Kernel =====  
  
1.0     1.2     -1.3    4.5  
-5.0    3.0     3.5     6.0  
-8.9    12.0    23.2    12.0  
13.0    -14.0   -15.0    16.0  
  
===== Added padding =====  
  
0.0     0.0     0.0     0.0     0.0  
0.0     -3.0    -4.0     4.5     0.0  
0.0     6.0     7.8     12.0    0.0  
0.0     5.0     -0.5    12.0    0.0  
0.0     0.0     0.0     0.0     0.0  
  
===== Result =====  
  
530.46
```

Figure 4.5: Another Standard testcase

```
Tests > Test_4 > output_matrix.txt  
1    530.5
```

Figure 4.6: Output


```
===== Input =====  
  
3 4 0 2  
-3.0 -4 4.5 6 7.8 12 5 -0.5 12.0  
1 1.2 -1.3 4.5 -5.0 3 3.5 6 -8.9 12 23.2 12 13 -14 -15 16.0
```

Figure 4.7: Test where size of image is smaller than kernel's

```
Tests > Test_1 > output_matrix.txt  
1 Error: size not match
```

Figure 4.8: Output of test where size of image is smaller than kernel's

THE END.