Analysis of Advanced 2D Convolution in Image Processing by Using AVX and OpenMP

Nwe Zin Oo
Faculty of Information Technology Supporting and Maintenance,
University of Computer Studies, Hpa-an,
Myanmar
5910130015@email.psu.ac.th

***Abstract*— In the field of digital image processing, the image data are required to operate for some enhancement operations such as image filtering, image restoration, image transformation, and so on. The memory allocation of image data is presented in an array and executed by two-dimensional matrix-matrix multiplication. The convolution operation is an essential operation among many filtering algorithms (e.g. Gaussian filter). In this paper, the characteristics of different convolution methods have been examined, and the important role of convolution and advanced parallel version of 2D convolution to improve the performance and instructions per clock cycle on a multi-core architecture by using AVX and OpenMP. Generally, convolution is performed by sliding the kernel over the image and moving the left side direction of the kernel through all data of the input matrix until all values of the image have been calculated. This paper intends to utilize and analyze how 1D, 2D, and 3D convolutions work in image processing and the parallel versions of 1D and 2D convolutions using padding on CPUs are proposed.**

Keywords—1D, 2D, 3D, convolution, image processing, Gaussian, parallel, AVX, OpenMP

I. INTRODUCTION

When signal processing and analysis are performed, the output of the system for any input signal can be constructed by using convolution operation. Note that a signal can be one-dimensional or two-dimensional or higher-dimensional inputs in some applications. When kernels are used to slide over the convolution to retrieve valuable information from the convolution, different kernel sizes containing different patterns of numbers produce different output results after convolution. Normally, convolution is used to transform image applications such as blurring effect, image sharpening, increasing or decreasing the contrast level, edge detection, and so on. There are some general steps to perform convolution on an image: (1) matrix is inverted by flipping horizontally and vertically, (2) sliding the kernel over the image starting from the left corner, (3) performing multiplication and accumulating the corresponding elements and then adding them to obtain the value of the output pixel at that corresponding location, (4) implementing the steps (2) and (3) repeatedly until all values of the image have been calculated. In this paper, 1D convolution means convolution of a one-dimensional signal, 2D convolution refers that the combination of 1D convolutions by convolving both directions horizontally and vertically, and 3D convolution operates where the kernel slides in three dimensions (height, width, channels of the image) as opposed to two dimensions with 2D convolutions. 3D convolution is more expensive in computation and has less memory efficiency since a three-dimensional matrix needs more memory space in the computer to preserve temporal storage and needs more calculations than 1D and 2D convolutional operations. The analysis of 1D, 2D, and 3D convolution processing is examined to knowledge of the comparisons between them. Some previous papers [1][2][3] reported on convolution in the dimensions of 1D and 2D and their advantages and usages in image processing filtering. [4] revealed with the high-performance implementation of 3D Convolutional Neural Networks on GPU processors and [5] also pointed out the accelerating of 2D and 3D convolution methods in FGPA. Moreover, the authors of [6] proposed a new 3D convolutional neural network for the application of human action recognition. Many approaches and researchers aim to focus on reducing the computational works and the way of measuring performance related to convolution, there has still been a remaining fact to analyze the comparisons between performance and power consumptions of different convolution methods such as 1D, 2D, and 3D convolutions on CPUs. This paper comprises all in one of the details of how 1D, 2D, and 3D convolutions work in image processing. Due to intensive multiplications with three kernels (R,G,B) and memory requirements becoming the most intensive work when using 3D convolution, it will leave as an extending work for the future. For 1D and 2D convolutions, this paper re-analyzes this gap by using AVX and OpenMP methods.

II. CONVOLUTION

Convolution is a fundamental operation to many common image processing that performs image enhancement by determining the value of a central pixel using additions the weighted values of all its neighbors together [3]. The size of the convolutional matrix can be determined by the size of a kernel which is important to determine the transformation effect of the convolution process by convoluting the input matrix and producing the output result. This section explains the details of 1D, 2D, and 3D convolution methods in image processing. To easily understand the flow of convolution, the Gaussian filter will be used as an example explanation in this paper.

A. 1D Convolution

First, the major advantage of 1D convolution is that a real-time and low-cost hardware implementation is feasible due to the simple and compact configuration of one-dimensional matrix multiplication that performs only 1D convolutions [7]. Unlike 2D convolution, there has only one dimension of a matrix to multiply and accumulate the values with one dimension of a kernel filter matrix to produce a new output value in the corresponding place.

There have some benefits of using 1D convolution are: (i) simple matrix multiplication by a kernel (ii) suitable for real-time fault detection and monitoring (iii) low-cost hardware implementation, and (iv) the number of multiplications and additions of 1D convolution is less than 2D and 3D convolutions.

2023 27th International Computer Science and Engineering Conference (ICSEC) | 979-8-3503-4210-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSEC59635.2023.10329733

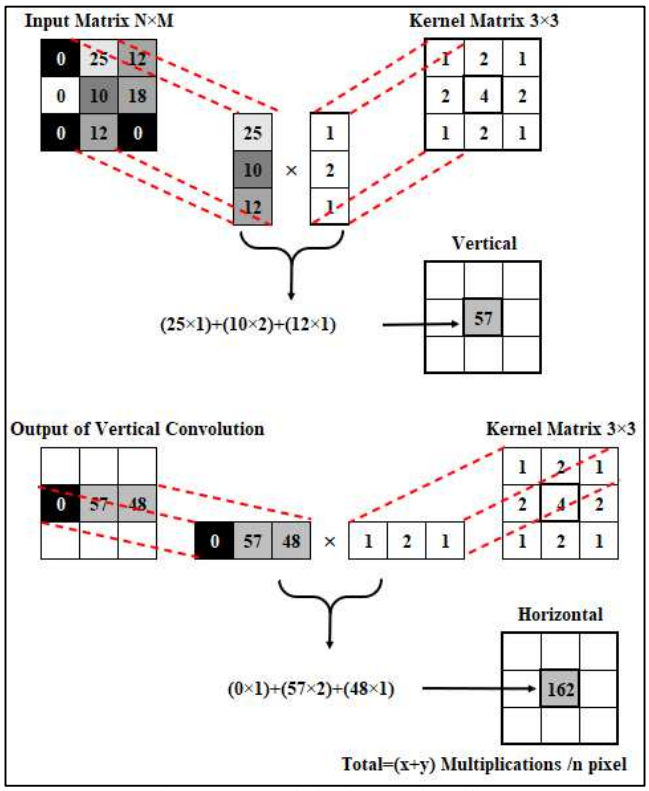

Fig. 1. 1D Convolution in Gaussian Filter.

Let z[n] be the output result of convolution (*) by the input x[2,3,4] and the filter y[4,2]. Therefore, each value of z[0], z[1], z[2], z[3], ..., can be calculated by using the following equation.

$$z[n] = \sum_{k=-\infty}^{\infty} x[k] \times y[n-k] \quad (1)$$

For example,

$$z[0] = \sum_{k=-\infty}^{\infty} x[k] \times y[0-k]$$
$$= (x[0]\times y[0])+(x[1]\times y[0\text{-}1])+(x[2]\times y[0\text{-}2])+ \ldots$$
$$= (x[0]\times y[0])$$
$$= (2 \times 4) = 8$$

$$z[1] = \sum_{k=-\infty}^{\infty} x[k] \times y[1-k]$$
$$=(x[0]\times y[1\text{-}0])+(x[1]\times y[1\text{-}1])+(x[2]\times y[1\text{-}2])+ \ldots$$
$$= (x[0]\times y[1]) + (x[1]\times y[0])$$
$$= (2 \times 2) + (3 \times 4) = 16$$

The other computations of z[0], z[1], z[2], z[3], ..., can be calculated using the above Eq.(1). After computing 1D convolution, adding these four outputs (z[0], z[1], z[2], z[3]) produces the output signal z[n] = {8, 16, 19, 16}.

In Gaussian, the input matrix is symmetric in order and the size of the filter kernel is odd. The kernel values are computed by using Eq. (1) for 1D Gaussian filtering.

$$G1(x) = \left(1/\sqrt{2 \times \pi\sigma}\right) \times e^{-\frac{x^2}{2\sigma^2}} \quad (2)$$

where, $e^{-\frac{x^2}{2\sigma^2}}$ is the Fourier transform function and σ is the standard deviation of the Gaussian function. Fig.1 explains the 1D convolution in the Gaussian filter.

B. 2D Convolution

In computer vision and deep learning, the 2D convolution method is very useful and the kernel convolves the input by moving the filters along the input vertically and horizontally to produce the convoluted 2D output. 2D Convolution filtering can be used to process sharpening effects, smoothing effects, edge detection, and texture analysis. One of the famous filters in image processing is a Gaussian filter which is a linear filter that can smooth an image and reduce undesired noises. The following Eq. (3) is an equation to execute 2D Gaussian filtering.

$$G2(x) = \left(1/\sqrt{2 \times \pi\sigma}\right) \times e^{-\frac{(x^2+y^2)}{2\sigma^2}} \quad (3)$$

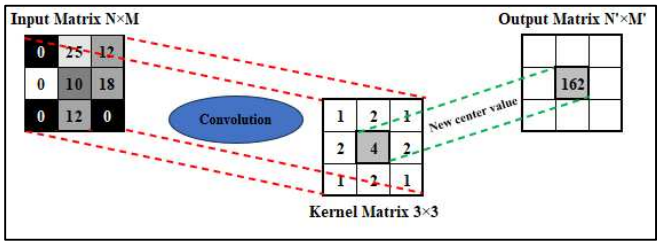

Fig. 2. 2D Convolution in Gaussian Filter.

Let z[m,n] be the output result of convolution (*) by the input x[m,n] and the filter y[m,n]. Therefore, each value of z[0,0], z[0,1], z[0,2], z[1,0], z[1,1], z[1,2], z[2,0], z[2,1], z[2,2], can be calculated by using the following equation.

$$z[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j] \times y[m-i, n-j] \quad (4)$$

For example,

$$z[1,1] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j] \times y[1-i, 1-j]$$
$$= x[0,0]\times y[1,1] + x[1,0]\times y[0,1] + x[2,0]\times y[-1,1] +$$
$$x[0,1]\times y[1,0] + x[1,1]\times y[0,0] + x[2,1]\times y[-1,0] +$$
$$x[0,2]\times y[1,-1] + x[1,2]\times y[0,-1] + x[2,2]\times y[-1,-1]$$
$$= 0\times 0 + 25\times 2 + 12\times 1 + 0\times 2 + 10\times 4 + 18\times 2 + 0\times 1 +$$
$$12\times 2 + 0\times 1$$
$$=162$$

C. 3D Convolution

In 3D convolution, a 3D filter can move in three directions (height, width, and channels of the image). At each corresponding position, the multiplication and addition are computed per element data that provide a new value of the output matrix. Since the filter slides through a 3D layer in the right-side direction via convolution, the output image pixels are arranged in a 3D format as well. Therefore, the final output matrix becomes a 3D data format. For example, color images can be commonly split into three channels (red/green/blue) for requiring 3D convolutional kernels. Such a 3D format is important for some applications, such as in 3D segmentations/reconstructions of biomedical imagining, e.g.,

CT scan images and MRI images where the important paths such as blood vessels meander around in the 3D space [6]. Fig. 3 and Eq. 5 reveal the underlying how 3D convolution has been done in color images.

$$z[m,n,r] = \sum_{k=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j,k] \times y[m-i, n-j, r-k] \tag{5}$$

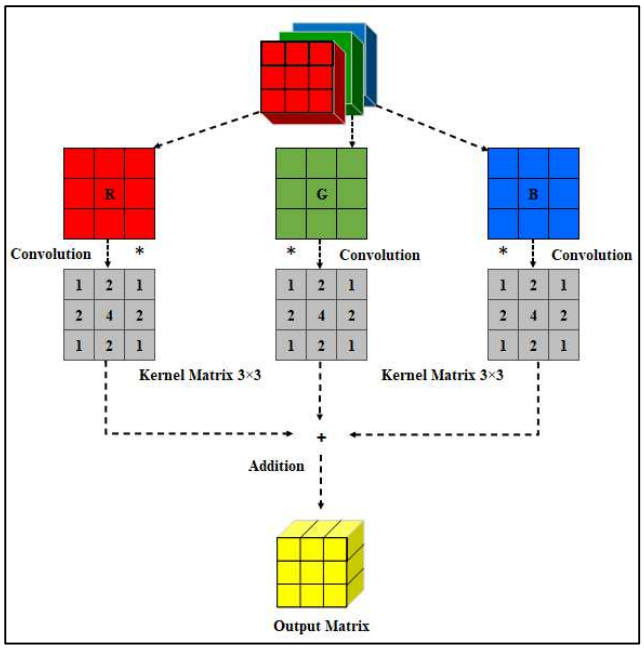

Fig. 3. 3D Convolution in Gaussian Filter.

At that point, 3D convolution is more expensive in computation and has less memory efficiency since a three-dimensional matrix needs more memory space in the computer to preserve temporal storage and needs more calculations than 1D and 2D convolutional operations. For this reason, it will be a future extension work.

III. Parallel Implementation

Most image applications use convolution for image filtering, noise reduction, smoothing, blurring, edge enhancement, and so on. To analyze the computational speed of different convolution methods (1D, 2D, and 3D), the input matrix and the kernel matrix should be the same size with the same data. The most common kernel (3×3) is applied for convolution comparisons. Since 1D convolution needs only the multiplication of one-dimensional arrays and the requirement of memory storage is the least than 2D and 3D convolution [9]. In 1D, **six** multiplications and **four** additions are required to obtain the corresponding value in the output matrix. On the other hand, 2D convolution requires **nine** multiplications and **six** additions. For this reason, an advanced version of 2D convolution using AVX and OpenMP is contributed to compare with the others. The (9×3)=27 multiplications with three kernels (R,G,B) and memory requirement are the most intensive work when using 3D convolution. Therefore, a parallel version of 3D convolution will not be included in this paper.

A. Parallel 1D Convolution

1D convolution is used in the area of audio and text data extraction, signal smoothing, and sentence classification. The convolutional kernel/filter has only one direction array to calculate the output which also becomes a 1D array [8].

There have advantages of using 1D convolution are:

- Reduction of dimensions and simple computations
- Efficient low memory space and feature pooling
- Applying separable convolution.

```
void 1D_Cov(float * input, float * kernel, float * dest){
__m256i c0 = _mm256_set1_epi16(kernel[0]);
__m256i c1 = _mm256_set1_epi16(kernel[1]);
__m256i c2 = _mm256_set1_epi16(kernel[2]);
#pragma omp parallel for schedule (static) private(i,j)

for (i=0; i<MAX1; i+=2){
  for (j=1; j<MAX1; j+=16){
       // AVX_256 intrinsic codes
/*The partial multiplied and added results are stored in
 temp1 and temp2, respectively */
 temp1 = _mm256_add_epi16(_mm256_add_epi16(
           _mm256_mullo_epi16(vec_10, c0),
           _mm256_mullo_epi16(vec_11, c1)),
           _mm256_mullo_epi16(vec_12, c2));
 // AVX_256 intrinsic codes for temp2
_mm256_storeu_si256( (__m256i *) &temp1[i][j], temp1);
_mm256_storeu_si256( (__m256i *) &temp2[i+1][j],temp2);
     }
}
/*Broadcasts 16-bit integer a to all elements of returned
  vector*/
// AVX_256 intrinsic codes
#pragma omp parallel for schedule (static) private (i,j)
for (i=1; i<MAX1; i+=2){
  for (j=0; j<MAX1; j+=16){
  /*Load 256-bits of integer data from memory into result_1
    and result_2, simultaneously*/
    dest_1 = _mm256_add_epi16(
              _mm256_add_epi16(
              _mm256_mullo_epi16(t_10, r0),
              _mm256_mullo_epi16(t_11, r1)),
              _mm256_mullo_epi16(t_12, r2));
// AVX_256 intrinsic codes for dest_2
/* Stores 256-bits of integer data from a into memory location
    of dest[i][j] and dest[i+1][j] simultaneously.*/
_mm256_storeu_si256((__m256i *) &dest[i][j], dest_1);
_mm256_storeu_si256((__m256i *) &dest[i+1][j], dest_2);
     } }
}
```

Fig. 4. Parallel 1D Convolution in Gaussian Filter [10].

The parallel version of the 1D Gaussian filter is shown in Fig. 4 by applying Eq. (2). This version is implemented by utilizing AVX_256 intrinsic instructions and OpenMP multi-threading. [10] also proposed the 1D Gaussian filter to enhance performance and reduce energy consumption. Fig. 1 shows the sequential version of the 1D Gaussian filter which concerns using Eq. (1) and Eq. (2) to obtain a single value in a convoluted matrix. Since the Gaussian filter can work separably vertically and horizontally it can reduce the number of multiplications instead of nine multiplications when the 3×3 kernel is multiplied for it.

B. Parallel 2D Convolution

2D convolution is famous for many applications of image processing, such as image blurring, smoothing, sharpening, edge detection of images, and so on. In Fig. 2, the center point of a kernel is y[0, 0] is **4** since the kernel value is located at the middle (centermost) of the kernel. For example, if the kernel size is 5×5, the indices of a kernel of 5 elements will be -2, -1, 0, 1, and 2. To easily understand the flow of 2D convolution, suppose that 3×3 input and 3×3 kernel matrices are convoluted as shown in Fig. 2.

There are three factors that need to consider for 2D convolution.

- Kernel size: The kernel size defines the dimensions of the convolution.
- Flipping the kernel: The kernel matrix is flipped both horizontally and vertically before multiplying through overlapped input matrix since x[0,0] is multiplied by the last sample of impulse response, y[1,1]. And x[2,2] is multiplied by the first value, y[-1,-1], where y[-1,-1] is assumed that out-of-axis means 0.
- Padding: To overcome out-of-axis issues, the zero padding method can be applied in convolution by padding 0 around the boundaries of the input matrix if necessary to define how the border of an image is allocated.

In multi-core architecture, CPUs can run more than one thread with independent instructions per core (the independent processor unit). On the other hand, most GPUs come with three different types of processing cores: CUDA, tensor, and ray-tracing cores. As a limitation, this paper only focuses on multi-core architecture with SIMD instructions on CPUs, the implementation and testing on GPUs processors will be becoming an extension for future work.

```
void 2D_Cov(float *input, float* kernel, float  *output, int
        temp_X, int temp_Y, int Hk_size, int Vk_size)
{
// find center position of kernel (half of kernel size)
          int center_X = (Hk_size - 1) / 2;
          int center_Y = (Vk_size - 1) / 2;
          int x_pad = (Hk_size / 2; // for padding
          int y_pad = (Vk_size / 2;

          int kernel_size = Hk_size * Vk_size;
          int next = (2 * x_pad + temp_X);

// allocate padding
float *padding = (float*)_aligned_malloc(sizeof(float)*
                        next*(2 * y_pad + temp_X), 16);

#pragma omp parallel for schedule(static)
    for (int i = 0; i<y_pad; i++)
      for (int j = 0; j<temp_X + (2 * x_pad); j++) {
       padding [j + next * i] = 0;
       padding [j + next*(temp_Y + (2 * y_pad) - i - 1)] = 0;
          }
```

(a)

```
#pragma omp parallel for schedule(static)
          for (int i = 0; i<temp_Y; i++) {
             for (int k = 0; k<x_pad; k++) {
          // flipped X axis and Y axis
                    }
           for (int j = 0; j<temp_X; j++) {
          // flipped X axis and Y axis
                    }
#pragma omp parallel for
{
 float* Flip_kernel = (float*)_aligned_malloc(sizeof(int)*
                        Xkernel*Ykernel, 16);
    for (int i = 0; i<kernel_size; i++) // kernel rows
          // row index of flipped kernel
          Flip_kernel[i] = kernel[kernel_size - i - 1];

#pragma omp parallel for
    for (int y = 0; y < temp_Y; y++) {
      for (int x = 0; x < temp_X; x += 8) {
          out_vec = _mm256_setzero_ps();
          for (int i = 0; i<(kernel_size / 9)*9 ; i += 9)
{
  /* To obtain 9 output values, the operations of addition and
  multiplication with Flip_kernel[i],[i+1],[i+2], ...,[i+8] are
  broadcasted and accumulated according to Xkernel size */
}

/* Store  outputs in the buffer tempOut */
_mm256_storeu_ps(&tempOut[x + (y)* temp_X], out_vec);
                    }
for (int x = temp_X; x< temp_X; x++, value = 0) {
  for (int i = 0; i<kernel_size; i++) {
  /* The final result is kept in the value variable after
     accumulating the results from Flip_kernel[i] */

   tempOut[x + y * temp_X] = value;
                    }
}}}
```

(b)

Fig. 5 (a) Padding and (b) Flipping the axis for parallel 2D Convolution.

IV. Performance Comparisons

In this section, computation time is compared to analyze which case achieves the best performance between parallel 1D and parallel 2D convolutions. The measurement of performance in terms of GFLOPS can be calculated to analyze the advantages and disadvantages of them. The tested environment contains a laptop computer with the specifications of *Intel core i7 8750H CPU@2.2 GHz*, a *Coffee Lake* 8th generation, *9MB* for *L3* cache, comprising of six cores and twelve threads with optimization (*-O2*). The simple kernels containing the same values are (3×3), (5×5), and (7×7). Fig. 6, Fig.7, and Fig. 8 are the experimental results after testing on CPUs for different matrix sizes of 1024, 2048 and 4096 by utilizing AVX_256 intransics and OpenMP.

This paper mainly focuses on data and task parallelism, therefore Advanced Vactor Extensions (AVX) [11] is the best option for microprocessor from Intel and AMD computers. Since AVX supports 256-bit instruction sets and fused multiply-add (FMA) instruction which improves face detection, image processing and high-performance computing, floating-point operations can be computed on CPUs embedded with 32 or 64 bits registers to exploit the computation of multiple data elements simultaneously and independently. OpenMP is an API supports shared memory parallel programming in C/C++ and Fortran. By announcing a

header file *#include(omp.h)* which is used for loop-level and function-level parallel programming [12], the execution time can be reduced by creating multi-threading to distribute loads and stores in parallel.

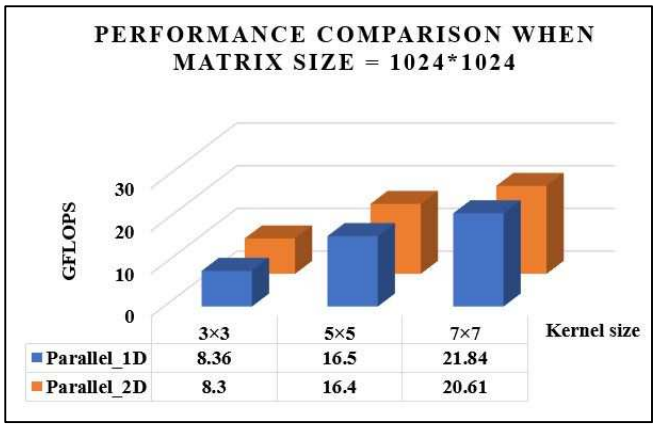

Fig. 6 Performance comparison between parallel versions of 1D and 2D convolution when matrix size is 1024×1024.

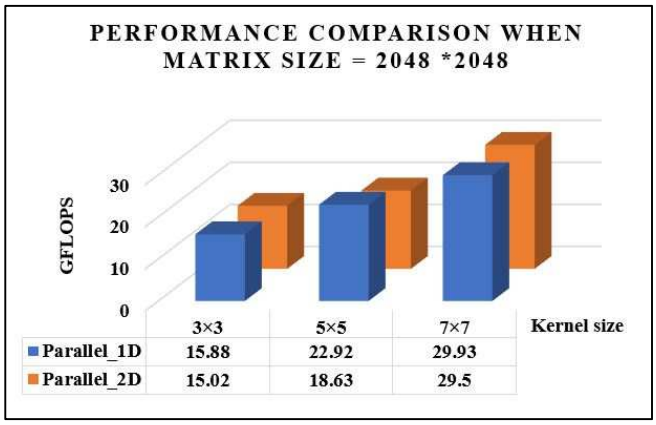

Fig. 7 Performance comparison between parallel versions of 1D and 2D convolution when matrix size is 2048×2048.

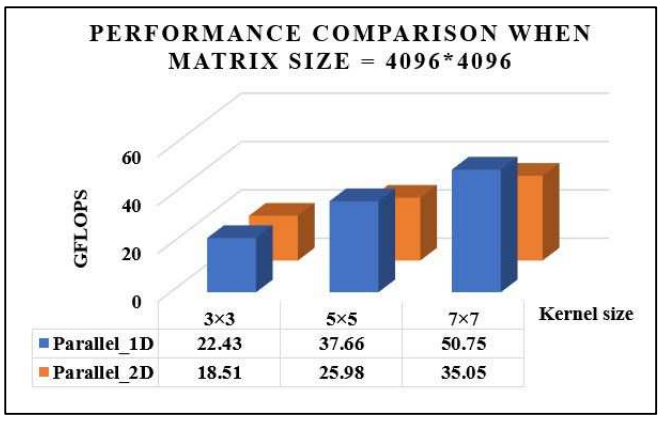

Fig. 8 Performance comparison between parallel versions of 1D and 2D convolution when matrix size is 4096×4096.

Based on the experimental results, a parallel version of 1D convolution outperforms a parallel version of 2D as shown in the figures. When the input matrix becomes large, the execution time for convolution, padding the matrix, flipping the kernel, and shifting the right side until all the pixels of an image have been multiplied are the reason why parallel 1D convolution achieves better performance than 2D convolution. According to tests, 1D achieves more 1.2 times to 1.4 times than 2D convolution in performance comparison. That is due to the number of multiplications and additions of 1D convolution is less than 2D convolution. Even though 1D convolution is slightly faster than 2D convolution, the proposed parallel version of 2D convolution can fulfill the requirement of high-performance matrix-matrix multiplication than the other methods. Moreover, in the proposed advanced 2D convolution, the zero-padding method is comprised of parallel implementation utilizing AVX for data parallelism and OpenMP for task parallelism.

V. Conclusion

The proposed two parallel versions applied the same approaches on CPU, the proposed parallel 2D convolution is very useful for image processing such as image filtering, blurring, smoothing, edge detection, and color enhancement. This paper intends to analyze the important role of the convolution method and to propose the advanced parallel version of 2D convolution by using AVX and OpenMP. Moreover, the advantages and disadvantages, of different characteristics of different convolution methods (1D, 2D and 3D) can be observed in this proposed paper when further research methodologies focus on parallel image processing on CPUs.

References

[1] Kiranyaz, S., Avcı, O., Abdeljaber, O., Ince, T., Gabbouj, M., & Inman, D.J, "1D Convolutional Neural Networks and Applications: A Survey", Computer Science, 2019.

[2] Chaudhary, N., Misra, S., Kalamkar, D.D., Heinecke, A., Georganas, E., Ziv, B., Adelman, M., & Kaul, B., "Efficient and Generic 1D Dilated Convolution Layer for Deep Learning", 2021.

[3] Kelefouras, Vasilios I. and Georgios Keramidas. "Design and Implementation of 2D Convolution on x86/x64 Processors." IEEE Transactions on Parallel and Distributed Systems PP (2022): 1-1.

[4] Lan, Qiang & Wang, Zelong & Wen, Mei & Zhang, Chun-yuan & Wang, Yijie, "High Performance Implementation of 3D Convolutional Neural Networks on a GPU", Computational Intelligence and Neuroscience, pp.1-8, 2017.

[5] Liu, Zhiqiang & Chow, Paul & Xu, Jinwei & Jiang, Jingfei & Dou, Yong & Zhou, Jie, "A Uniform Architecture Design for Accelerating 2D and 3D CNNs on FPGAs" Electronics. 8(1), 65, 2019.

[6] Ji S., Xu W., Yang M., Yu K, "3D Convolutional neural networks for human action recognition", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol:35(1): pp. 221–231, 2013.

[7] Kiranyaz, Serkan & Avci, Onur & Abdeljaber, Osama & Ince, Turker & Gabbouj, Moncef & Inman, Daniel, "1D Convolutional Neural Networks and Applications: A Survey", Mechanical Systems and Signal Processing. 151, April 2021.

[8] Tousimojarad, A., Vanderbauwhede, W., & Cockshott, W.P, "2D Image Convolution using Three Parallel Programming Models on the Xeon Phi", 2017.

[9] Andrade-Ambriz, Y.A., Ledesma, S., & Almanza-Ojeda, D.L., "Multithreading Programming for Feature Extraction in Digital Images", Advances in Intelligent Systems and Computing, 2019.

[10] N.Zin Oo, "The Improvement of 1D Gaussian Blur Filter using AVX and OpenMP", 2022 22nd International Conference on Control, Automation and Systems (ICCAS), pp. 1493-1496, 2022.

[11] Intel® Advanced Vector Extensions Intrinsics, (Online). Retrieved on August 8, 2023, https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide/and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions/details-of-intel-advanced-vector-extensionsintrinsics.html.

[12] Barbara M. Chapman, Gabriele Jost, Ruud van der Pas, "Using OpenMP: Portable Shared Memory Parallel Programming", pp. 23-68, United States of America, MIT Press, 2008.