DAFTAR PUSTAKA

Abdollahzadeh, M. (2022). *Implementing Deep Convolutional Neural Networks in C without External Libraries*. Medium. https://towardsdatascience.com/implementing-deep-convolutional-neural-networks-in-c-without-external-libraries-b30464f64d02

Abiodun, O. I., Jantan, A., Omolara, A. E., Dada, K. V., Mohamed, N. A., & Arshad, H. (2018). *State-of-the-art in artificial neural network applications: A survey*. *Heliyon*, *4*(11), e00938. https://doi.org/10.1016/j.heliyon.2018.e00938

Agrawal, V., & Danaher, J. (2015). *Performance of a Computer*. Auburn University.

Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J., Fadhel, M. A., Al-Amidie, M., & Farhan, L. (2021). *Review of deep learning: concepts, CNN architectures, challenges, applications, future directions*. *Journal of Big Data*, *8*(1). https://doi.org/10.1186/s40537-021-00444-8

Barney, Frederick, Livermore Computing, & LLNL. (n.d.). *Introduction to Parallel Computing Tutorial | HPC @ LLNL*. Diakses tanggal 6 September 2023, dari https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial##LimitsCosts

Chervyakov, N., Lyakhov, P., & Nagornov, N. (2020). *Analysis of the Quantization Noise in Discrete Wavelet Transform Filters for 3D Medical Imaging*. *Applied Sciences*, *10*(4), 1223. https://doi.org/10.3390/app10041223

Choi RY, Coyner AS, Kalpathy-Cramer J, Chiang MF, Campbell JP. *Introduction to machine learning, neural networks, and deep learning*. Trans Vis Sci Tech. 2020;9(2):14, https://doi.org/10.1167/tvst.9.2.14

Dexon Systems. (2022). *What are RGB and YUV color spaces?*. DEXON Systems. Diakses tanggal 22 September 2024, dari https://dexonsystems.com/blog/rgb-yuv-color-spaces

Dong, C. (2014). *Image Super-Resolution Using Deep Convolutional Networks*. arXiv.org. https://arxiv.org/abs/1501.00092

Dong, C. (2016). *Accelerating the Super-Resolution Convolutional Neural Network*. arXiv.org. https://arxiv.org/abs/1608.00367

Eddelbuettel, D. (2020). *Parallel computing with R: A brief review*. *WIREs Computational Statistics*, *13*(2). https://doi.org/10.1002/wics.1515

ffmpeg. (n.d.). *ffmpeg Documentation*. https://www.*ffmpeg*.org/. Diakses tanggal 12 Oktober 2023, dari https://www.*ffmpeg*.org/*ffmpeg*-all.html#psnr

Hore, A., & Ziou, D. (2010, August). *Image quality metrics: PSNR vs. SSIM. In 2010 20th international conference on pattern recognition* (pp. 2366-2369). *IEEE*.

Indolia, S., Goswami, A. K., Mishra, S., & Asopa, P. (2018). *Conceptual Understanding of Convolutional Neural Network- A Deep Learning Approach*. *Procedia Computer Science*, *132*, 679–688. https://doi.org/10.1016/j.procs.2018.05.069

Intel, & Gillespie. (n.d.). *Amdahl's Law, Gustafson's Trend, and the Performance Limits of Parallel Applications*. *Intel*. Diakses tanggal 29 Oktober 2023, dari https://www.intel.com/content/dam/develop/external/us/en/documents/gillespie-0053-aad-gustafson-amdahl-v1-2-rh-final-145023.pdf

Jadon, Shruti; Yadav, Rama Shankar (2016). [IEEE 2016 11th International Conference on Industrial and Information Systems (ICIIS) - Roorkee, India (2016.12.3-2016.12.4)] 2016 11th International Conference on Industrial and Information Systems (ICIIS) - Multicore processor: Internal structure, architecture, issues, challenges, scheduling strategies and performance. , (), 381–386. doi:10.1109/ICIINFS.2016.8262970

Khan, G. (2017). *Computer Performance*. Ryerson University

Microsoft. (2022). Tentang YUV Video - Win32 apps. Microsoft Learn. Diakses tanggal 22 September 2023 , https://learn.microsoft.com/id-id/windows/win32/medfound/about-yuv-video

OpenMP Architecture Review Board. (2023). *OpenMP*. Wikipedia. Diakses tanggal 6 September 2023, dari https://en.wikipedia.org/wiki/OpenMP

Pires, Paulo & Santos, José & Pereira, Inês. (2023). *Artificial Neural Networks: History and State of the Art*. 10.4018/978-1-6684-7366-5.ch037.

Sethi, A. & Kushwah, H. (2015). Multicore processor technology-advantages and challenges. International Journal of Research in Engineering and Technology, 4(09), 87-89.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2014). *Operating System Concepts Essentials, Binder Ready Version*. Wiley.

Solomon, D. (2007). *Data Compression*. *Springer Science & Business Media*.

Lampiran 1 Source code Program Serial YUV Video Super-Resolution

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>


void FSRCNN(double *img_hr, double *img_lr, int rows, int cols, int scale);

void imfilter(double *img, double *kernel, double *img_fltr, int rows, int cols, int
padsize);

void pad_image(double *img, double *img_pad, int rows, int cols, int padsize);

void PReLU(double *img_fltr, int rows, int cols, double bias, double prelu_coeff);

double Max(double a, double b);

double Min(double a, double b);

void imadd(double *img_fltr_crnt, double *img_fltr_prev, int cols, int rows);

void deconv(double *img_input, double *img_output, double *kernel, int cols, int
rows, int stride);

void double_2_uint8(double *double_img, unsigned char *uint8_img, int cols, int
rows);

double cpu_time_accumulation;

time_t wall_time_accumulation;

clock_t
s1,s1_end,sum_s1,s2,s2_end,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s3_end,s4_end,s
5_end,s6_end,s7_end,s8_end,s9_end,s10_end,s11_end,s12_end;

clock_t
p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p1_end,p2_end,p3_end,p4_end,p5_e
nd,p6_end,p7_end,p8_end,p9_end,p10_end,p11_end,p12_end,p13_end;

clock_t fconv,fconv_end,fprelu, fprelu_end, fimadd, fimadd_end;

double
sum_fcnt=0,p1_sum=0,p2_sum=0,p3_sum=0,p4_sum=0,p5_sum=0,p6_sum=0,p7
_sum=0,p8_sum=0,p9_sum=0,p10_sum=0,sum_conv_1=0.0,sum_conv_2=0.0,su
m_conv_3=0.0,sum_conv_4=0.0,sum_conv_5=0.0,sum_conv_6=0.0,sum_conv_7
=0.0,sum_prelu1=0.0,sum_prelu2=0.0,sum_prelu3=0.0,sum_prelu4=0.0,sum_prel
u5=0.0,sum_prelu6=0.0,sum_prelu7=0.0,sum_imadd2=0.0,sum_imadd3=0.0,sum
```

```
_imadd4=0.0,sum_imadd5=0.0,sum_imadd6=0.0,sum_imadd7=0.0,sum_imadd8=
0.0;

int main(int argc, char *argv[])

{

        char *inFile = argv[1];

        char *outFile = argv[2];


        //Upsampler parameters

        int scale = 2;


        //Compressed Assault Cube

        int num;

        printf("Masukkan jumlah total frame video yang akan diupscale : ");

        scanf("%d",&num);


        int inCols = 176; //Width of input (downsampled) video

        int inRows = 144; //Height of input (downsampled) video


        int outCols = inCols*scale;

        int outRows = inRows*scale;



        FILE *inFp, *outFp;

        //things that are used to measure cpu and wall time

        //cpu time

        clock_t begin = clock();

        //wall time
```

```
        time_t start, ending;
    time(&start);
        s1= clock();
        inFp = fopen(inFile, "rb");
        if (inFp == NULL)
        {
          printf("\n We have null pointer \n");
        }
        outFp = fopen(outFile, "wb");
        if (outFp == NULL)
        {
          printf("\n We have null pointer \n");
        }

        // To read and write each frame in an unsigned character format
        unsigned char *inBuf = (unsigned char *)malloc(inCols*inRows*sizeof(unsigned char));
        unsigned char *outBuf = (unsigned char *)malloc(outCols*outRows*sizeof(unsigned char));
        // To work with each pixel in the range of 0~1
        double *inBuf_tmp = (double *)malloc(inCols*inRows*sizeof(double));
        double *outBuf_tmp = (double *)malloc(outCols*outRows*sizeof(double));
        s1_end= clock();
        sum_fcnt += (double)(s1_end - s1)/CLOCKS_PER_SEC;

        for (int fcnt = 0; fcnt < num; fcnt++)
        {
```

```
//////// Interpolate each frame using FSRCNN for Y component and simple repitition for U and V components

// Pointer to obtain value of each tpixel of input frame

s2= clock();

unsigned char *inP = inBuf;

double *inP_tmp = inBuf_tmp;

// Pointer to obtain value of each pixel of output frame

unsigned char *outP = outBuf;

double *outP_tmp = outBuf_tmp;


//Y Component

fread(inBuf, sizeof(unsigned char), inCols*inRows, inFp);

int i, j;


for (i = 0; i<inRows; i++)

for (j = 0; j<inCols; j++)

{

        int cnt = i*inCols + j;

        int x = *inP++;

        *(inP_tmp + cnt) = (double)(x / 255.0);

}

s2_end= clock();

sum_fcnt += (double)(s2_end-s2)/CLOCKS_PER_SEC;

FSRCNN(outP_tmp, inP_tmp, inRows, inCols, scale);

s3= clock();

outP_tmp = outBuf_tmp;
```

```
for (i = 0; i<inRows*scale; i++)
for (j = 0; j<inCols*scale; j++)
{
        int cnt = i*inCols*scale + j;
        *(outP_tmp + cnt) = *(outP_tmp + cnt) * 255;
}


double_2_uint8( outP_tmp, outP, outCols, outRows);

fwrite(outBuf, sizeof(unsigned char), outCols*outRows, outFp);

//U Component
fread(inBuf, sizeof(unsigned char), inCols*inRows / 4, inFp);

inP = inBuf;
outP = outBuf;

for (i = 0; i < inRows / 2; i++)
for (j = 0; j < inCols / 2; j++) {

        int cnt = 2 * (i * outCols / 2 + j);

        unsigned char x = *inP++;

        *(outP + cnt) = x;
        *(outP + cnt + 1) = x;
```

```
            *(outP + cnt + outCols / 2) = x;
            *(outP + cnt + outCols / 2 + 1) = x;

}

fwrite(outBuf, sizeof(unsigned char), outCols*outRows / 4, outFp);

// V COmponent
fread(inBuf, sizeof(unsigned char), inCols*inRows / 4, inFp);
inP = inBuf;
outP= outBuf;

for (i = 0; i < inRows / 2; i++)
for (j = 0; j < inCols / 2; j++) {

            int cnt = 2 * (i*outCols / 2 + j);

            unsigned char x = *inP++;

            *(outP + cnt) = x;
            *(outP + cnt + 1) = x;
            *(outP + cnt + outCols / 2) = x;
            *(outP + cnt + outCols / 2 + 1) = x;

}

fwrite(outBuf, sizeof(unsigned char), outCols*outRows / 4, outFp);
```

```
  s3_end= clock();
  sum_fcnt += (double)(s3_end-s3)/CLOCKS_PER_SEC;
}
//get cpu time end
clock_t end = clock();
//get wall time end
time(&ending);
// calculate cpu and wall time
double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
time_t wall_time = ending - start;
//print out cpu and wall time
printf("CPU time spent to upscale the video is %lf seconds", time_spent);
printf("\nWall time spent to upscale the video is %f seconds\n", wall_time);
printf("\nTime spent on running serial fraction is : %lf seconds\n",sum_fcnt);
printf("\nTime spent on running parallel fraction for layer 1 is : %f seconds\n",p1_sum);
printf("\nTime spent on running parallel fraction for layer 2 is : %f seconds\n",p2_sum);
printf("\nTime spent on running parallel fraction for layer 3 is : %f seconds\n",p3_sum);
printf("\nTime spent on running parallel fraction for layer 4 is : %f seconds\n",p4_sum);
printf("\nTime spent on running parallel fraction for layer 5 is : %f seconds\n",p5_sum);
printf("\nTime spent on running parallel fraction for layer 6 is : %f seconds\n",p6_sum);
printf("\nTime spent on running parallel fraction for layer 7 is : %f seconds\n",p7_sum);
free(inBuf);
inBuf = NULL;
```

```
        free(inBuf_tmp);
        inBuf_tmp = NULL;
        free(outBuf);
        outBuf = NULL;
        free(outBuf_tmp);
        outBuf_tmp = NULL;
}

void FSRCNN(double *img_hr, double *img_lr, int rows, int cols, int scale)
{
        // General Settings
        s4= clock();
        int num_layers = 8;
        /////////// Convolution1 -------- Layer1
        // Reading weights of first layer
        FILE *weights_layer1_ptr;
        weights_layer1_ptr = fopen("weights_layer1.txt", "r");
        if (weights_layer1_ptr == NULL) { printf("Error in the reading weights of
first layer\n"); };
        double weights_layer1[1400];
        for (int i = 0; i < 1400; i++)
        {
          fscanf(weights_layer1_ptr, "%lf", &weights_layer1[i]);
          //printf("%lf\n", weights_layer1[i]);
        }
        fclose(weights_layer1_ptr);
        // Reading biases of first layer
```

```
FILE *biases_layer1_ptr;

biases_layer1_ptr = fopen("biasess_layer1.txt", "r");

if (biases_layer1_ptr == NULL) { printf("Error in the reading biases of first layer\n"); };

double biases_layer1[56];

for (int i = 0; i < 56; i++)

{

  fscanf(biases_layer1_ptr, "%lf", &biases_layer1[i]);

}

fclose(biases_layer1_ptr);

// other parameters

int filtersize = 25; //5X5

int patchsize = 5;

int padsize = (patchsize - 1) / 2;

int num_filters = 56;

double prelu_coeff_layer1 = -0.8986;


// Convolution

double *img_fltr_1 = (double *)malloc(rows * cols * num_filters * sizeof(double));

double *kernel = (double *)malloc(filtersize*sizeof(double));


double *img_fltr_p1 = img_fltr_1; // Pointer to img_fltr1 ==>> Using this way to be able to shift it to access data


int cnt_weight = 0;


double bias_tmp;
```

```
    s4_end= clock();
    sum_fcnt += (double)(s4_end-s4)/CLOCKS_PER_SEC;
    //cpu time
    // clock_t layer_begin= clock();
    // //wall time
    // time_t layer_start, layer_ending;
// time(&layer_start);
    p1 = clock();
    for (int i = 0; i < num_filters; i++)
    {
      // reading corresponding weights to kernel pointer
      for (int cnt_kernel = 0; cnt_kernel < filtersize; cnt_kernel++)
      {
              *(kernel + cnt_kernel) = weights_layer1[cnt_weight + cnt_kernel];
      }
      cnt_weight = cnt_weight + filtersize;

      imfilter(img_lr, kernel, img_fltr_p1, rows, cols, padsize);



      bias_tmp = biases_layer1[i];

      PReLU(img_fltr_p1, rows, cols, bias_tmp, prelu_coeff_layer1);

      img_fltr_p1 = img_fltr_p1 + cols*rows;
    }
    p1_end = clock();
```

```
p1_sum += (double)( p1_end - p1 )/CLOCKS_PER_SEC;

////////// Convolution2 ---------- Layer 2~7


////////// Layer2

// Reading weights of 2nd layer

s5= clock();

FILE *weights_layer2_ptr;

weights_layer2_ptr = fopen("weights_layer2.txt", "r");

if (weights_layer2_ptr == NULL) { printf("Error in the reading weights of 2nd layer\n"); };

// Note: weights must be saved in a way which that corresponding weights of each channel can be read by pointer concept ==>> for this layer 12X56 matrix is reshaped to (12X56)*1 vector

double weights_layer2[672];

for (int i = 0; i < 672; i++)

{

  fscanf(weights_layer2_ptr, "%lf", &weights_layer2[i]);

}

fclose(weights_layer2_ptr);

// Reading biases of 2nd layer

FILE *biases_layer2_ptr;

biases_layer2_ptr = fopen("biasess_layer2.txt", "r");

if (biases_layer2_ptr == NULL) { printf("Error in the reading biases of 2nd layer\n"); };

double biases_layer2[12];

for (int i = 0; i < 12; i++)

{

  fscanf(biases_layer2_ptr, "%lf", &biases_layer2[i]);

}
```

```
fclose(biases_layer2_ptr);

// Other parameters

int filtersize2 = 1; //1X1

int patchsize2 = 1;

int padsize2 = (patchsize2 - 1) / 2;

int num_filters2 = 12;

int num_channels2 = 56;

double prelu_coeff_layer2 = 0.3236;

// Convolution

double *img_fltr_2 = (double *)calloc(rows * cols * num_filters2 , sizeof(double)); // use calloc to initialize all variables to zero

double *img_fltr_2_tmp = (double *)malloc(rows * cols * sizeof(double));

double *kernel2 = (double *)malloc(filtersize2*sizeof(double));

double *img_fltr_p2 = img_fltr_2; // Pointer to img_fltr2




cnt_weight = 0;

s5_end= clock();

sum_fcnt += (double)(s5_end-s5)/CLOCKS_PER_SEC;

p2 = clock();

for (int i = 0; i < num_filters2; i++)

{

  img_fltr_p1 = img_fltr_1; // Return pointer to the first of array which contains feature map of previous layer

  for (int j = 0; j < num_channels2; j++)

  {

        // reading corresponding weights to kernel

        for (int cnt_kernel = 0; cnt_kernel < filtersize2; cnt_kernel++)
```

```
                {
                        *(kernel2 + cnt_kernel) = weights_layer2[cnt_weight +
cnt_kernel];
                }

                imfilter(img_fltr_p1, kernel2, img_fltr_2_tmp, rows, cols,
padsize2);

                imadd(img_fltr_p2, img_fltr_2_tmp, cols, rows);



                cnt_weight = cnt_weight + filtersize2;
                img_fltr_p1 = img_fltr_p1 + rows*cols;
          }
          bias_tmp = biases_layer2[i];

          PReLU(img_fltr_p2, rows, cols, bias_tmp, prelu_coeff_layer2);



          img_fltr_p2 = img_fltr_p2 + rows*cols;
        }
        p2_end = clock();
        p2_sum += (double)(p2_end-p2)/CLOCKS_PER_SEC;
        s6= clock();
        free(img_fltr_1);
        img_fltr_1 = NULL;
        free(kernel);
        kernel = NULL;
```

```
free(img_fltr_2_tmp);

img_fltr_2_tmp = NULL;


////////// Layer3

// Reading weights of 3rd layer

FILE *weights_layer3_ptr;

weights_layer3_ptr = fopen("weights_layer3.txt", "r");

if (weights_layer3_ptr == NULL) { printf("Error in the reading weights of 3rd layer\n"); };

double weights_layer3[1296];

for (int i = 0; i < 1296; i++)

{

  fscanf(weights_layer3_ptr, "%lf", &weights_layer3[i]);

}

fclose(weights_layer3_ptr);

// Reading biases of 3rd layer

FILE *biases_layer3_ptr;

biases_layer3_ptr = fopen("biasess_layer3.txt", "r");

if (biases_layer3_ptr == NULL) { printf("Error in the reading biases of 3rd layer\n"); };

double biases_layer3[12];

for (int i = 0; i < 12; i++)

{

  fscanf(biases_layer3_ptr, "%lf", &biases_layer3[i]);

}

fclose(biases_layer3_ptr);

// Other parameters

int filtersize3 = 9; //3X3
```

```
int patchsize3 = 3;

int padsize3 = (patchsize3 - 1) / 2;

int num_filters3 = 12;

int num_channels3 = 12;

double prelu_coeff_layer3 = 0.2288;

// Convolution

double *img_fltr_3 = (double *)calloc(rows * cols * num_filters3 , sizeof(double));

double *kernel3 = (double *)malloc(filtersize3*sizeof(double));

double *img_fltr_p3 = img_fltr_3; // Pointer to img_fltr2

double *img_fltr_3_tmp = (double *)malloc(rows * cols * sizeof(double));


cnt_weight = 0;

s6_end= clock();

sum_fcnt += (double)(s6_end-s6)/CLOCKS_PER_SEC;

p3 = clock();

for (int i = 0; i < num_filters3; i++)

{

  img_fltr_p2 = img_fltr_2; // Return pointer to the first cell of array which contains feature maps of previous layer

  for (int j = 0; j < num_channels3; j++)

  {

        // reading corresponding weights to kernel

        for (int cnt_kernel = 0; cnt_kernel < filtersize3; cnt_kernel++)

        {

              *(kernel3 + cnt_kernel) = weights_layer3[cnt_weight + cnt_kernel];

        }
```

```
                imfilter(img_fltr_p2, kernel3, img_fltr_3_tmp, rows, cols,
padsize3);

                imadd(img_fltr_p3, img_fltr_3_tmp, cols, rows);

                cnt_weight = cnt_weight + filtersize3;
                img_fltr_p2 = img_fltr_p2 + rows*cols;
            }
            bias_tmp = biases_layer3[i];

            PReLU(img_fltr_p3, rows, cols, bias_tmp, prelu_coeff_layer3);

            img_fltr_p3 = img_fltr_p3 + rows*cols;
        }
        p3_end = clock();
        p3_sum += (double)(p3_end-p3)/CLOCKS_PER_SEC;
        s7= clock();
        free(img_fltr_2);
        img_fltr_2 = NULL;
        free(img_fltr_2_tmp);
        img_fltr_2_tmp = NULL;
        free(kernel2);
        kernel2 = NULL;
        free(img_fltr_3_tmp);
        img_fltr_3_tmp = NULL;
```

```
////////// Layer4

// Reading weights of 4th layer

FILE *weights_layer4_ptr;

weights_layer4_ptr = fopen("weights_layer4.txt", "r");

if (weights_layer4_ptr == NULL) { printf("Error in the reading weights of 4th layer\n"); };

double weights_layer4[1296];

for (int i = 0; i < 1296; i++)

{

  fscanf(weights_layer4_ptr, "%lf", &weights_layer4[i]);

}

fclose(weights_layer4_ptr);

// Reading biases of 4th layer

FILE *biases_layer4_ptr;

biases_layer4_ptr = fopen("biasess_layer4.txt", "r");

if (biases_layer4_ptr == NULL) { printf("Error in the reading biases of 4th layer\n"); };

double biases_layer4[12];

for (int i = 0; i < 12; i++)

{

  fscanf(biases_layer4_ptr, "%lf", &biases_layer4[i]);

}

fclose(biases_layer4_ptr);

// Other parameters

int filtersize4 = 9; //3X3

int patchsize4 = 3;

int padsize4 = (patchsize4 - 1) / 2;

int num_filters4 = 12;
```

```
int num_channels4 = 12;

double prelu_coeff_layer4 = 0.2476;

// Convolution

double *img_fltr_4 = (double *)calloc(rows * cols * num_filters4 , sizeof(double));

double *kernel4 = (double *)malloc(filtersize4*sizeof(double));

double *img_fltr_p4 = img_fltr_4; // Pointer to img_fltr4

double *img_fltr_4_tmp = (double *)malloc(rows * cols * sizeof(double));


cnt_weight = 0;

s7_end= clock();

sum_fcnt += (double)(s7_end-s7)/CLOCKS_PER_SEC;

p4 = clock();

for (int i = 0; i < num_filters4; i++)

{

  img_fltr_p3 = img_fltr_3;

  for (int j = 0; j < num_channels4; j++)

  {

        // reading corresponding weights to kernel

        for (int cnt_kernel = 0; cnt_kernel < filtersize4; cnt_kernel++)

        {

              *(kernel4 + cnt_kernel) = weights_layer4[cnt_weight + cnt_kernel];

        }


        imfilter(img_fltr_p3, kernel4, img_fltr_4_tmp, rows, cols, padsize4);
```

```
            imadd(img_fltr_p4, img_fltr_4_tmp, cols, rows);


            cnt_weight = cnt_weight + filtersize4;
            img_fltr_p3 = img_fltr_p3 + rows*cols;
  }
  bias_tmp = biases_layer4[i];


  PReLU(img_fltr_p4, rows, cols, bias_tmp, prelu_coeff_layer4);


  img_fltr_p4 = img_fltr_p4 + rows*cols;
}
p4_end = clock();
p4_sum += (double)(p4_end-p4)/CLOCKS_PER_SEC;
s8= clock();
free(img_fltr_3);
img_fltr_3 = NULL;
free(img_fltr_3_tmp);
img_fltr_3_tmp = NULL;
free(kernel3);
kernel3 = NULL;
free(img_fltr_4_tmp);
img_fltr_4_tmp = NULL;


////////// Layer5
// Reading weights of 5th layer
FILE *weights_layer5_ptr;
weights_layer5_ptr = fopen("weights_layer5.txt", "r");
```

```
if (weights_layer5_ptr == NULL) { printf("Error in the reading weights of 5th layer\n"); };

double weights_layer5[1296];

for (int i = 0; i < 1296; i++)

{

  fscanf(weights_layer5_ptr, "%lf", &weights_layer5[i]);

}

fclose(weights_layer5_ptr);

// Reading biases of 5th layer

FILE *biases_layer5_ptr;

biases_layer5_ptr = fopen("biasess_layer5.txt", "r");

if (biases_layer5_ptr == NULL) { printf("Error in the reading biases of 5th layer\n"); };

double biases_layer5[12];

for (int i = 0; i < 12; i++)

{

  fscanf(biases_layer5_ptr, "%lf", &biases_layer5[i]);

}

fclose(biases_layer5_ptr);

// Other parameters

int filtersize5 = 9; //3X3

int patchsize5 = 3;

int padsize5 = (patchsize5 - 1) / 2;

int num_filters5 = 12;

int num_channels5 = 12;

double prelu_coeff_layer5 = 0.3495;

// Convolution
```

```
double *img_fltr_5 = (double *)calloc(rows * cols * num_filters5 , sizeof(double));

double *kernel5 = (double *)malloc(filtersize5*sizeof(double));

double *img_fltr_p5 = img_fltr_5; // Pointer to img_fltr5

double *img_fltr_5_tmp = (double *)malloc(rows * cols * sizeof(double));


cnt_weight = 0;

s8_end= clock();

sum_fcnt += (double)(s8_end-s8)/CLOCKS_PER_SEC;

p5 = clock();

for (int i = 0; i < num_filters5; i++)

{

  img_fltr_p4 = img_fltr_4;

  for (int j = 0; j < num_channels5; j++)

  {

        // reading corresponding weights to kernel

        for (int cnt_kernel = 0; cnt_kernel < filtersize5; cnt_kernel++)

        {

              *(kernel5 + cnt_kernel) = weights_layer5[cnt_weight + cnt_kernel];

        }


        imfilter(img_fltr_p4, kernel5, img_fltr_5_tmp, rows, cols, padsize5);


        imadd(img_fltr_p5, img_fltr_5_tmp, cols, rows);
```

```
            cnt_weight = cnt_weight + filtersize5;
            img_fltr_p4 = img_fltr_p4 + rows*cols;
  }
  bias_tmp = biases_layer5[i];
  // fprelu = clock();
  PReLU(img_fltr_p5, rows, cols, bias_tmp, prelu_coeff_layer5);
  // fprelu_end = clock();
  // sum_prelu5 += (double)(fprelu_end-fprelu)/CLOCKS_PER_SEC;
  img_fltr_p5 = img_fltr_p5 + rows*cols;
}
p5_end = clock();
p5_sum += (double)(p5_end-p5)/CLOCKS_PER_SEC;
s9= clock();
free(img_fltr_4);
img_fltr_4 = NULL;
free(img_fltr_4_tmp);
img_fltr_4_tmp = NULL;
free(kernel4);
kernel4 = NULL;
free(img_fltr_5_tmp);
img_fltr_5_tmp = NULL;

////////// Layer6
// Reading weights of 6th layer
FILE *weights_layer6_ptr;
weights_layer6_ptr = fopen("weights_layer6.txt", "r");
```

```
if (weights_layer6_ptr == NULL) { printf("Error in the reading weights of 6th layer\n"); };

double weights_layer6[1296];

for (int i = 0; i < 1296; i++)

{

  fscanf(weights_layer6_ptr, "%lf", &weights_layer6[i]);

}

fclose(weights_layer6_ptr);

// Reading biases of 6th layer

FILE *biases_layer6_ptr;

biases_layer6_ptr = fopen("biasess_layer6.txt", "r");

if (biases_layer6_ptr == NULL) { printf("Error in the reading biases of 6th layer\n"); };

double biases_layer6[12];

for (int i = 0; i < 12; i++)

{

  fscanf(biases_layer6_ptr, "%lf", &biases_layer6[i]);

}

fclose(biases_layer6_ptr);

// Other parameters

int filtersize6 = 9; //3X3

int patchsize6 = 3;

int padsize6 = (patchsize6 - 1) / 2;

int num_filters6 = 12;

int num_channels6 = 12;

double prelu_coeff_layer6 = 0.7806;

// Convolution
```

```
        double *img_fltr_6 = (double *)calloc(rows * cols * num_filters6 , sizeof(double));

        double *kernel6 = (double *)malloc(filtersize6*sizeof(double));

        double *img_fltr_p6 = img_fltr_6; // Pointer to img_fltr6

        double *img_fltr_6_tmp = (double *)malloc(rows * cols * sizeof(double));


        cnt_weight = 0;

        s9_end= clock();

        sum_fcnt += (double)(s9_end-s9)/CLOCKS_PER_SEC;

        p6 = clock();

        for (int i = 0; i < num_filters6; i++)

        {

          img_fltr_p5 = img_fltr_5;

          for (int j = 0; j < num_channels6; j++)

          {

                  // reading corresponding weights to kernel

                  for (int cnt_kernel = 0; cnt_kernel < filtersize6; cnt_kernel++)

                  {

                          *(kernel6 + cnt_kernel) = weights_layer6[cnt_weight + cnt_kernel];

                  }


                  imfilter(img_fltr_p5, kernel6, img_fltr_6_tmp, rows, cols, padsize6);


                  imadd(img_fltr_p6, img_fltr_6_tmp, cols, rows);
```

```
            cnt_weight = cnt_weight + filtersize6;
            img_fltr_p5 = img_fltr_p5 + rows*cols;
        }
        bias_tmp = biases_layer6[i];

        PReLU(img_fltr_p6, rows, cols, bias_tmp, prelu_coeff_layer6);

        img_fltr_p6 = img_fltr_p6 + rows*cols;
      }
      p6_end = clock();
      p6_sum += (double)(p6_end-p6)/CLOCKS_PER_SEC;
      s10= clock();
      free(img_fltr_5);
      img_fltr_5 = NULL;
      free(img_fltr_5_tmp);
      img_fltr_5_tmp = NULL;
      free(kernel5);
      kernel5 = NULL;

      ////////// Layer7
      // Reading weights of 7th layer
      FILE *weights_layer7_ptr;
      weights_layer7_ptr = fopen("weights_layer7.txt", "r");
      if (weights_layer7_ptr == NULL) { printf("Error in the reading weights of 7th layer\n"); };
      double weights_layer7[672];
      for (int i = 0; i < 672; i++)
```

```
{
  fscanf(weights_layer7_ptr, "%lf", &weights_layer7[i]);
}
fclose(weights_layer7_ptr);
// Reading biases of 7th layer
FILE *biases_layer7_ptr;
biases_layer7_ptr = fopen("biasess_layer7.txt", "r");
if (biases_layer7_ptr == NULL) { printf("Error in the reading biases of 7th layer\n"); };
double biases_layer7[56];
for (int i = 0; i < 56; i++)
{
  fscanf(biases_layer7_ptr, "%lf", &biases_layer7[i]);
}
fclose(biases_layer7_ptr);
// Other parameters
int filtersize7 = 1; //1X1
int patchsize7 = 1;
int padsize7 = (patchsize7 - 1) / 2;
int num_filters7 = 56;
int num_channels7 = 12;
double prelu_coeff_layer7 = 0.0087;
// Convolution
double *img_fltr_7 = (double *)calloc(rows * cols * num_filters7 , sizeof(double));
double *kernel7 = (double *)malloc(filtersize7*sizeof(double));
double *img_fltr_p7 = img_fltr_7; // Pointer to img_fltr7
double *img_fltr_7_tmp = (double *)malloc(rows * cols * sizeof(double));
```

```
cnt_weight = 0;
s10_end= clock();
sum_fcnt += (double)(s10_end-s10)/CLOCKS_PER_SEC;
p7 = clock();
for (int i = 0; i < num_filters7; i++)
{
  img_fltr_p6 = img_fltr_6;
  for (int j = 0; j < num_channels7; j++)
  {
        // reading corresponding weights to kernel
        for (int cnt_kernel = 0; cnt_kernel < filtersize7; cnt_kernel++)
        {
              *(kernel7 + cnt_kernel) = weights_layer7[cnt_weight + cnt_kernel];
        }

        imfilter(img_fltr_p6, kernel7, img_fltr_7_tmp, rows, cols, padsize7);

        imadd(img_fltr_p7, img_fltr_7_tmp, cols, rows);

        cnt_weight = cnt_weight + filtersize7;
        img_fltr_p6 = img_fltr_p6 + rows*cols;
  }
  bias_tmp = biases_layer7[i];
  fprelu = clock();
  PReLU(img_fltr_p7, rows, cols, bias_tmp, prelu_coeff_layer7);
```

```
            fprelu_end = clock();
            sum_prelu7 += (double)(fprelu_end-fprelu)/CLOCKS_PER_SEC;
            img_fltr_p7 = img_fltr_p7 + rows*cols;
        }
        p7_end = clock();
        p7_sum += (double)(p7_end-p7)/CLOCKS_PER_SEC;

        // //get cpu time end
        // clock_t layer_end = clock();
        // //get wall time end
        // time(&layer_ending);
        // calculate cpu and wall time
        // cpu_time_accumulation += (double)(layer_end - layer_begin) / CLOCKS_PER_SEC;
        // wall_time_accumulation += layer_ending - layer_start;
        s11= clock();
        free(img_fltr_6);
        img_fltr_6 = NULL;
        free(img_fltr_6_tmp);
        img_fltr_6_tmp = NULL;
        free(kernel6);
        kernel6 = NULL;

        /////////// Convolution3 ---------- Layer 8

        /////////// Layer8
        // Reading weights of 8th layer
```

```
FILE *weights_layer8_ptr;

weights_layer8_ptr = fopen("weights_layer8.txt", "r");

if (weights_layer8_ptr == NULL) { printf("Error in the reading weights of 8th layer\n"); };

double weights_layer8[4536];

for (int i = 0; i < 4536; i++)

{

  fscanf(weights_layer8_ptr, "%lf", &weights_layer8[i]);

}

fclose(weights_layer8_ptr);

// Reading biases of 8th layer

double biases_layer8 = - 0.03262640000;

// Other parameters

int filtersize8 = 81; //9x9

int patchsize8 = 9;

int num_filters8 = 1;

int num_channels8 = 56;


// Decvolution ==> output is the img_hr

double *img_fltr_8 = (double *)calloc((rows*scale) *(cols*scale) * num_filters8 , sizeof(double));

double *kernel8 = (double *)malloc(filtersize8*sizeof(double));

double *img_fltr_8_tmp = (double *)malloc((rows*scale) *(cols*scale) * sizeof(double));


cnt_weight = 0;

img_fltr_p7 = img_fltr_7;
```

```
        for (int j = 0; j < num_channels8; j++)
        {
          // reading corresponding weights to kernel
          for (int cnt_kernel = 0; cnt_kernel < filtersize8; cnt_kernel++)
          {
                *(kernel8 + cnt_kernel) = weights_layer8[cnt_weight + cnt_kernel];
          }
          cnt_weight = cnt_weight + filtersize8;

          deconv(img_fltr_p7, img_fltr_8_tmp, kernel8, cols, rows, scale);
          fimadd = clock();
          imadd(img_fltr_8, img_fltr_8_tmp, cols*scale, rows*scale);
          fimadd_end = clock();
          sum_imadd8 += (double)(fimadd_end-fimadd)/CLOCKS_PER_SEC;
          img_fltr_p7 = img_fltr_p7 + rows*cols;
        }


        for (int i=0;i<rows*scale;i++)
        for (int j = 0;j<cols*scale; j++)
        {
          int cnt_fnl = i*cols*scale + j;
          *(img_hr + cnt_fnl) = *(img_fltr_8 + cnt_fnl) + biases_layer8;
        }

        /*for ( int i = 0; i < 10; i++)
```

```
        {
          printf("%f\n", *(img_hr + i));
        }*/

        free(img_fltr_7);
        img_fltr_7 = NULL;
        free(img_fltr_7_tmp);
        img_fltr_7_tmp = NULL;
        free(kernel7);
        kernel7 = NULL;

        free(img_fltr_8);
        img_fltr_8 = NULL;
        free(img_fltr_8_tmp);
        img_fltr_8_tmp = NULL;
        free(kernel8);
        kernel8 = NULL;
        s11_end= clock();
        sum_fcnt += (double)(s11_end-s11)/CLOCKS_PER_SEC;
}


void imfilter(double *img, double *kernel, double *img_fltr, int rows, int cols, int
padsize)
{
        // img_pad is the pointer to padded image
        // kernel is the pointer to the kernel which used for convolution
```

```
// img_fltr is the pointer to the filtered image by applying convolution

// s12= clock();

int cols_pad = cols + 2 * padsize;

int rows_pad = rows + 2 * padsize;

int i, j, cnt, cnt_pad, cnt_krnl, k1, k2;

double sum;


double *img_pad = (double *)malloc(rows_pad * cols_pad * sizeof(double));

pad_image(img, img_pad, rows, cols, padsize);

// s12_end= clock();

// sum_fcnt += (double)(s12_end-s12)/CLOCKS_PER_SEC;

// p8 = clock();

for (i = padsize; i < rows_pad - padsize; i++)

for (j = padsize; j < cols_pad - padsize; j++)

{

  cnt = (i - padsize)*cols + (j - padsize); // counter which shows current pixel in filtered image (central pixel in convolution window)

  sum = 0;

  cnt_krnl = 0; // counter which determines kernel elements

  for (k1 = -padsize; k1 <= padsize; k1++)

  for (k2 = -padsize; k2 <= padsize; k2++)

  {

        cnt_pad = (i + k1)*cols_pad + j + k2; // counter which shows each neighbouring pixel of padded image used for convolution with kernel

        sum = sum + (*(img_pad + cnt_pad))*(*(kernel + cnt_krnl));

        cnt_krnl++;

  }
```

```
            *(img_fltr + cnt) = sum;
        }
        // p8_end = clock();
        // p8_sum += (double)(p8_end-p8)/CLOCKS_PER_SEC;

        free(img_pad);
        img_pad = NULL;
}

// Replicate image padding by the factor of "padsize"
void pad_image(double *img, double *img_pad, int rows, int cols, int padsize)
{ // This function receives an image and paddes its border in a replicative manner
        int cols_pad = cols + 2 * padsize;
        int rows_pad = rows + 2 * padsize;
        int i, j, k, cnt, cnt_pad, k1, k2;
        // Centeral pixels
        for (i = padsize; i < rows_pad - padsize; i++)
        for (j = padsize; j < cols_pad - padsize; j++)
        {
          cnt_pad = i * cols_pad + j;
          cnt = (i - padsize)*(cols) + j - padsize;
          double x = *(img + cnt);
          *(img_pad + cnt_pad) = x;
        }
        // Top and Bottom Rows
        for (j = padsize; j < cols_pad - padsize; j++)
        for (k = 0; k < padsize; k++)
```

```
{
  // Top Rows
  cnt_pad = j + k*cols_pad;
  cnt = j - padsize;
  *(img_pad + cnt_pad) = *(img + cnt);
  // Bottom Rows
  cnt_pad = j + (rows_pad - 1 - k)* cols_pad;
  cnt = (j - padsize) + (rows - 1)*cols;
  *(img_pad + cnt_pad) = *(img + cnt);
}
// Left and Right Columns
for (i = padsize; i < rows_pad - padsize; i++)
for (k = 0; k < padsize; k++)
{
  // Left Columns
  cnt = (i - padsize)*cols;
  cnt_pad = i*cols_pad + k;
  *(img_pad + cnt_pad) = *(img + cnt);
  // Right Columns
  cnt = (i - padsize)*cols + cols - 1;
  cnt_pad = i*cols_pad + cols_pad - 1 - k;
  *(img_pad + cnt_pad) = *(img + cnt);
}
// Corner Pixels
for (k1 = 0; k1 < padsize; k1++)
for (k2 = 0; k2 < padsize; k2++)
{
```

```
            // Upper Left Corner
            cnt_pad = k1*cols_pad + k2;
            *(img_pad + cnt_pad) = *(img);
            // Upper Right Corner
            cnt_pad = k1*cols_pad + cols_pad - 1 - k2;
            *(img_pad + cnt_pad) = *(img + cols - 1);
            // Lower Left Corner
            cnt_pad = (rows_pad - 1 - k1)*cols_pad + k2;
            *(img_pad + cnt_pad) = *(img + (rows - 1)*cols);
            // Lower Right Corner
            cnt_pad = (rows_pad - 1 - k1)*cols_pad + cols_pad - 1 - k2;
            *(img_pad + cnt_pad) = *(img + (rows - 1)*cols + cols - 1);
        }
}

void PReLU(double *img_fltr,int rows, int cols, double bias, double prelu_coeff)
{
        int cnt = 0;
        // p9 = clock();
        for (int i = 0; i < rows;i++)
        for (int j = 0; j < cols; j++)
        {
            cnt = i*cols + j;
            *(img_fltr + cnt) = Max(*(img_fltr + cnt) + bias, 0) + prelu_coeff *
Min(*(img_fltr + cnt) + bias, 0);
        }
        // p9_end = clock();
```

```
        // p9_sum += (double)(p9_end-p9)/CLOCKS_PER_SEC;
}

double Max(double a, double b)
{
        double c;
        c = a > b ? a : b;
        return c;
}

double Min(double a, double b)
{
        double c;
        c = a > b ? b : a;
        return c;
}

void imadd(double *img_fltr_sum, double *img_fltr_crnt, int cols, int rows)
{
        // *img_fltr_crnt ==> pointer to current feature map
        // *img_fltr_sum ==> pointer to the cumulutive feature map

        int cnt = 0;
        // p10 = clock();
        for (int i = 0; i < rows;i++)
        for (int j = 0; j < cols; j++)
        {
```

```
            cnt = i*cols + j;

            *(img_fltr_sum + cnt) = *(img_fltr_sum + cnt) + *(img_fltr_crnt + cnt);

        }

        // p10_end = clock();

        // p10_sum += (double)(p10_end-p10)/CLOCKS_PER_SEC;

}



void deconv(double *img_input, double *img_output, double *kernel, int cols, int
rows, int stride)

{

        int border = 1;

        int fsize = 9;

        int rows_pad = rows + 2 * border;

        int cols_pad = cols + 2 * border;

        double *img_input_padded = (double *)malloc(rows_pad * cols_pad *
sizeof(double));

        pad_image(img_input, img_input_padded, rows, cols, border);



        int rows_out_pad = rows_pad * stride;

        int cols_out_pad = cols_pad * stride;

        double *img_output_tmp = (double *)calloc((rows_out_pad + fsize - 1)*
(cols_out_pad + fsize - 1), sizeof(double));

        double *kernel_modif = (double *)malloc(fsize * fsize * sizeof(double));



        int idx, idy;

        for (int i = 0; i < rows_pad; i++)

        for (int j = 0; j < cols_pad; j++)
```

```
{
  int cnt_img = i*cols_pad + j;
  idx = i*stride;
  idy = j*stride;
  int cnt_img_output = idx*(cols_out_pad + fsize - 1) + idy; // (idx,idy) coordinate in temporal output image
  int cnt_kernel = 0;
  for (int k_r = 0; k_r < fsize; k_r++)
  {
  for (int k_c = 0; k_c < fsize; k_c++)
  {
        cnt_kernel = k_r*fsize + k_c;
        *(kernel_modif + cnt_kernel) = (*(kernel + cnt_kernel))*(*(img_input_padded + cnt_img));
        *(img_output_tmp + cnt_img_output + k_c) = *(img_output_tmp + cnt_img_output + k_c) + *(kernel_modif + cnt_kernel);

  }
  cnt_img_output = cnt_img_output + (cols_out_pad + fsize - 1);
   }

}

int rows_out = rows*stride;
int cols_out = cols*stride;

for (int i = 0; i < rows_out; i++)
for (int j = 0; j < cols_out; j++)
```

```
        {
          int i_tmp = i + ((fsize + 1) / 2) + stride*border - 1;
          int j_tmp = j + ((fsize + 1) / 2) + stride*border - 1;
          int cnt_img_out = i*cols_out + j;
          int cnt_img_out_tmp = i_tmp*(cols_out_pad + fsize - 1) + j_tmp; // (cols-pad+fsize-1) is the number of columns in the img_out_tmp
          *(img_output + cnt_img_out) = *(img_output_tmp + cnt_img_out_tmp);

        }

        free(img_input_padded); img_input_padded = NULL;
        free(img_output_tmp); img_output_tmp = NULL;
        free(kernel_modif); kernel_modif = NULL;
}

void double_2_uint8(double *double_img, unsigned char *uint8_img, int cols, int rows)
{
        int i, j, cnt, k;

        for (i = 0; i < rows;i++)
        for (j = 0; j < cols; j++)
        {
          cnt = i*cols + j;

          if (*(double_img + cnt) < 0)
                  * (uint8_img + cnt) = 0;
          if (*(double_img + cnt) > 255)
```

```
                * (uint8_img + cnt) = 255;


        for (k = 0; k < 255; k++)
        {
                if (*(double_img + cnt) >= k && *(double_img + cnt) < (k+0.5))
                *(uint8_img + cnt) =  k;


                if (*(double_img + cnt) >= (k+0.5) && *(double_img + cnt) <
(k+1))
                        *(uint8_img + cnt) = k + 1;
        }


      }
}
```

Lampiran 2 Source code Program Paralel YUV Video Super-Resolution

```
// This code is the C implementation of FSRCNN algorithm for YUV 4:2:0 video interpolation

// Milad Abdollahzadeh, 09/02/2017


#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <omp.h>


void FSRCNN(double *img_hr, double *img_lr, int rows, int cols, int scale);

void imfilter(double *img, double *kernel, double *img_fltr, int rows, int cols, int padsize);

void pad_image(double *img, double *img_pad, int rows, int cols, int padsize);

void PReLU(double *img_fltr, int rows, int cols, double bias, double prelu_coeff);

double Max(double a, double b);

double Min(double a, double b);

void imadd(double *img_fltr_crnt, double *img_fltr_prev, int cols, int rows);

void deconv(double *img_input, double *img_output, double *kernel, int cols, int rows, int stride);

void double_2_uint8(double *double_img, unsigned char *uint8_img, int cols, int rows);



int main(int argc, char *argv[])

{

	char *inFile = argv[1];

	char *outFile = argv[2];

	omp_set_num_threads(8);
```

```
    #pragma omp parallel
{

   printf("Hello World... from thread = %d\n",
        omp_get_thread_num());
}
    //Upsampler parameters
    int scale = 2;

    //Input jumlah frame yang akan diinterpolasikan
    int num;
    printf("Masukkan jumlah total frame video yang akan diupscale : ");
    scanf("%d",&num);

    //Compressed Assault Cube

    int inCols; //Width of input (downsampled) video
    int inRows; //Height of input (downsampled) video
    printf("Masukkan ukuran video\n");
    printf("Lebar video (Width)\t: ");
    scanf("%d",&inCols);
    printf("Tinggi video (Height)\t: ");
    scanf("%d",&inRows);

    int outCols = inCols*scale;
    int outRows = inRows*scale;
```

```
        FILE *inFp, *outFp;


        inFp = fopen(inFile, "rb");
        if (inFp == NULL)
        {
          printf("\n We have null pointer \n");
        }
        outFp = fopen(outFile, "wb");
        if (outFp == NULL)
        {
          printf("\n We have null pointer \n");
        }


        // To read and write each frame in an unsigned character format
        unsigned char *inBuf = (unsigned char *)malloc(inCols*inRows*sizeof(unsigned char));
        unsigned char *outBuf = (unsigned char *)malloc(outCols*outRows*sizeof(unsigned char));
        // To work with each pixel in the range of 0~1
        double *inBuf_tmp = (double *)malloc(inCols*inRows*sizeof(double));
        double *outBuf_tmp = (double *)malloc(outCols*outRows*sizeof(double));
        //things that are used to measure cpu and wall time
        //cpu time
        clock_t begin = clock();
        //wall time
        time_t start, ending;
    time(&start);
        //#pragma omp parallel for
```

```
for (int fcnt = 0; fcnt < num; fcnt++)
{
  //////// Interpolate each frame using FSRCNN for Y component and simple repitition for U and V components
  // Pointer to obtain value of each tpixel of input frame
  unsigned char *inP = inBuf;
  double *inP_tmp = inBuf_tmp;
  // Pointer to obtain value of each pixel of output frame
  unsigned char *outP = outBuf;
  double *outP_tmp = outBuf_tmp;

  //Y Component
  fread(inBuf, sizeof(unsigned char), inCols*inRows, inFp);
  int i, j;
  // #pragma omp parallel for
  for (i = 0; i<inRows; i++)
  for (j = 0; j<inCols; j++)
  {
        int cnt = i*inCols + j;
        int x = *inP++;
        *(inP_tmp + cnt) = (double)(x / 255.0);
  }
  // #pragma omp parallel for collapse(2)
 // for (i = 0; i < inRows; i++)
 // {
//  for (j = 0; j < inCols; j++)
//  {
```

```
//            int cnt = i * inCols + j;
//                        int x = *inP++;
//            *(inP_tmp + cnt) = (double)(x / 255.0);
//  }
 // }
   FSRCNN(outP_tmp, inP_tmp, inRows, inCols, scale);

   outP_tmp = outBuf_tmp;

   for (i = 0; i<inRows*scale; i++)
   for (j = 0; j<inCols*scale; j++)
   {
            int cnt = i*inCols*scale + j;
            *(outP_tmp + cnt) = *(outP_tmp + cnt) * 255;
   }


   double_2_uint8( outP_tmp, outP, outCols, outRows);

   fwrite(outBuf, sizeof(unsigned char), outCols*outRows, outFp);

   //U Component
   fread(inBuf, sizeof(unsigned char), inCols*inRows / 4, inFp);

   inP = inBuf;
   outP = outBuf;
```

```
for (i = 0; i < inRows / 2; i++)
for (j = 0; j < inCols / 2; j++) {

        int cnt = 2 * (i * outCols / 2 + j);

        unsigned char x = *inP++;

        *(outP + cnt) = x;
        *(outP + cnt + 1) = x;
        *(outP + cnt + outCols / 2) = x;
        *(outP + cnt + outCols / 2 + 1) = x;

}

fwrite(outBuf, sizeof(unsigned char), outCols*outRows / 4, outFp);

// V COmponent
fread(inBuf, sizeof(unsigned char), inCols*inRows / 4, inFp);
inP = inBuf;
outP= outBuf;

for (i = 0; i < inRows / 2; i++)
for (j = 0; j < inCols / 2; j++) {

        int cnt = 2 * (i*outCols / 2 + j);

        unsigned char x = *inP++;
```

```
            *(outP + cnt) = x;
            *(outP + cnt + 1) = x;
            *(outP + cnt + outCols / 2) = x;
            *(outP + cnt + outCols / 2 + 1) = x;

  }

  fwrite(outBuf, sizeof(unsigned char), outCols*outRows / 4, outFp);

}
//get cpu time end
clock_t end = clock();
//get wall time end
time(&ending);

// calculate cpu and wall time
double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
time_t wall_time = ending - start;
//print out cpu and wall time
printf("CPU time spent to upscale the video is %lf seconds", time_spent);
printf("\nWall time spent to upscale the video is %f seconds\n", wall_time);
free(inBuf);
inBuf = NULL;
free(inBuf_tmp);
inBuf_tmp = NULL;
free(outBuf);
```

```
        outBuf = NULL;
        free(outBuf_tmp);
        outBuf_tmp = NULL;
}

void FSRCNN(double *img_hr, double *img_lr, int rows, int cols, int scale)
{
        // General Settings
        int num_layers = 8;

        /////////// Convolution1 -------- Layer1
        // Reading weights of first layer
        FILE *weights_layer1_ptr;
        weights_layer1_ptr = fopen("weights_layer1.txt", "r");
        if (weights_layer1_ptr == NULL) { printf("Error in the reading weights of first layer\n"); };
        double weights_layer1[1400];
        for (int i = 0; i < 1400; i++)
        {
          fscanf(weights_layer1_ptr, "%lf", &weights_layer1[i]);
          //printf("%lf\n", weights_layer1[i]);
        }
        fclose(weights_layer1_ptr);
        // Reading biases of first layer
        FILE *biases_layer1_ptr;
        biases_layer1_ptr = fopen("biasess_layer1.txt", "r");
        if (biases_layer1_ptr == NULL) { printf("Error in the reading biases of first layer\n"); };
```

```
double biases_layer1[56];
for (int i = 0; i < 56; i++)
{
  fscanf(biases_layer1_ptr, "%lf", &biases_layer1[i]);
}
fclose(biases_layer1_ptr);
// other parameters
int filtersize = 25; //5X5
int patchsize = 5;
int padsize = (patchsize - 1) / 2;
int num_filters = 56;
double prelu_coeff_layer1 = -0.8986;


// Convolution
double *img_fltr_1 = (double *)malloc(rows * cols * num_filters * sizeof(double));
double *kernel = (double *)malloc(filtersize*sizeof(double));
double *img_fltr_p1 = img_fltr_1; // Pointer to img_fltr1 ==>> Using this way to be able to shift it to access data
#pragma omp parallel for
for (int i = 0; i < num_filters; i++)
{
  // reading corresponding weights to kernel pointer


  // for (int cnt_kernel = 0; cnt_kernel < filtersize; cnt_kernel++)
  // {
  //         *(kernel + cnt_kernel) = weights_layer1[cnt_weight + cnt_kernel];
  // }
```

```
        // cnt_weight = cnt_weight + filtersize;

        imfilter(img_lr, weights_layer1+i*filtersize, img_fltr_p1+i*cols*rows, rows, cols, padsize);

        PReLU(img_fltr_p1+i*cols*rows, rows, cols,biases_layer1[i], prelu_coeff_layer1);

        // img_fltr_p1 = img_fltr_p1 + cols*rows;
      }

      /////////// Convolution2 ---------- Layer 2~7

      /////////// Layer2
      // Reading weights of 2nd layer
      FILE *weights_layer2_ptr;
      weights_layer2_ptr = fopen("weights_layer2.txt", "r");
      if (weights_layer2_ptr == NULL) { printf("Error in the reading weights of 2nd layer\n"); };
      // Note: weights must be saved in a way which that corresponding weights of each channel can be read by pointer concept ==>> for this layer 12X56 matrix is reshaped to (12X56)*1 vector
      double weights_layer2[672];
      for (int i = 0; i < 672; i++)
      {
        fscanf(weights_layer2_ptr, "%lf", &weights_layer2[i]);
      }
      fclose(weights_layer2_ptr);
      // Reading biases of 2nd layer
```

```
FILE *biases_layer2_ptr;

biases_layer2_ptr = fopen("biasess_layer2.txt", "r");

if (biases_layer2_ptr == NULL) { printf("Error in the reading biases of 2nd layer\n"); };

double biases_layer2[12];

for (int i = 0; i < 12; i++)

{

  fscanf(biases_layer2_ptr, "%lf", &biases_layer2[i]);

}

fclose(biases_layer2_ptr);

// Other parameters

int filtersize2 = 1; //1X1

int patchsize2 = 1;

int padsize2 = (patchsize2 - 1) / 2;

int num_filters2 = 12;

int num_channels2 = 56;


double prelu_coeff_layer2 = 0.3236;

// Convolution

double *img_fltr_2 = (double *)calloc(rows * cols * num_filters2 , sizeof(double)); // use calloc to initialize all variables to zero

double *kernel2 = (double *)malloc(filtersize2*sizeof(double));

double *img_fltr_p2 = img_fltr_2;

double img_fltr_2_tmp[rows*cols];

#pragma omp parallel for firstprivate(img_fltr_2_tmp)

for (int i = 0; i < num_filters2; i++)

{

  for (int j = 0; j < num_channels2; j++)
```

```
        {

imfilter(img_fltr_1+j*cols*rows,weights_layer2+i*num_channels2+j,
img_fltr_2_tmp, rows, cols, padsize2);

                imadd(img_fltr_p2+i*cols*rows, img_fltr_2_tmp, cols, rows);

        }

        PReLU(img_fltr_p2+i*cols*rows, rows, cols, biases_layer2[i],
prelu_coeff_layer2);

    }


    free(img_fltr_1);

    img_fltr_1 = NULL;

    free(kernel);

    kernel = NULL;


    /////////// Layer3

    // Reading weights of 3rd layer

    FILE *weights_layer3_ptr;

    weights_layer3_ptr = fopen("weights_layer3.txt", "r");

    if (weights_layer3_ptr == NULL) { printf("Error in the reading weights of
3rd layer\n"); };

    double weights_layer3[1296];

    for (int i = 0; i < 1296; i++)

    {

      fscanf(weights_layer3_ptr, "%lf", &weights_layer3[i]);

    }

    fclose(weights_layer3_ptr);

    // Reading biases of 3rd layer

    FILE *biases_layer3_ptr;
```

```
biases_layer3_ptr = fopen("biasess_layer3.txt", "r");

if (biases_layer3_ptr == NULL) { printf("Error in the reading biases of 3rd layer\n"); };

double biases_layer3[12];

for (int i = 0; i < 12; i++)

{

  fscanf(biases_layer3_ptr, "%lf", &biases_layer3[i]);

}

fclose(biases_layer3_ptr);

// Other parameters

int filtersize3 = 9; //3X3

int patchsize3 = 3;

int padsize3 = (patchsize3 - 1) / 2;

int num_filters3 = 12;

int num_channels3 = 12;

double prelu_coeff_layer3 = 0.2288;

// Convolution

double *img_fltr_3 = (double *)calloc(rows * cols * num_filters3 , sizeof(double));

double *kernel3 = (double *)malloc(filtersize3*sizeof(double));

double *img_fltr_p3 = img_fltr_3; // Pointer to img_fltr2

// double *img_fltr_3_tmp = (double *)malloc(rows * cols * sizeof(double));

img_fltr_p2 = img_fltr_2;

#pragma omp parallel for

for (int i = 0; i < num_filters3; i++)

{

  double img_fltr_3_tmp[rows*cols];
```

```
        for (int j = 0; j < num_channels3; j++)
        {
                // reading corresponding weights to kernel
                // for (int cnt_kernel = 0; cnt_kernel < filtersize3; cnt_kernel++)
                // {
                //      *(kernel3 + cnt_kernel) = weights_layer3[cnt_weight + cnt_kernel];
                //      // printf("index : %d\ttesting: %d\n",cnt_weight + cnt_kernel,i*num_channels3+j+filtersize3);
                // }

                imfilter(img_fltr_p2+j*cols*rows, weights_layer3+(i*num_channels3+j)*filtersize3 , img_fltr_3_tmp, rows, cols, padsize3);
                imadd(img_fltr_p3+i*cols*rows, img_fltr_3_tmp, cols, rows);

                // cnt_weight = cnt_weight + filtersize3;
                // img_fltr_p2 = img_fltr_p2 + rows*cols;
        }
        PReLU(img_fltr_p3+i*cols*rows, rows, cols, biases_layer3[i], prelu_coeff_layer3);
        // img_fltr_p3 = img_fltr_p3 + rows*cols;
    }

    free(img_fltr_2);
    img_fltr_2 = NULL;
    free(kernel2);
    kernel2 = NULL;
```

```
//////////// Layer4

// Reading weights of 4th layer

FILE *weights_layer4_ptr;

weights_layer4_ptr = fopen("weights_layer4.txt", "r");

if (weights_layer4_ptr == NULL) { printf("Error in the reading weights of 4th layer\n"); };

double weights_layer4[1296];

for (int i = 0; i < 1296; i++)

{

  fscanf(weights_layer4_ptr, "%lf", &weights_layer4[i]);

}

fclose(weights_layer4_ptr);

// Reading biases of 4th layer

FILE *biases_layer4_ptr;

biases_layer4_ptr = fopen("biasess_layer4.txt", "r");

if (biases_layer4_ptr == NULL) { printf("Error in the reading biases of 4th layer\n"); };

double biases_layer4[12];

for (int i = 0; i < 12; i++)

{

  fscanf(biases_layer4_ptr, "%lf", &biases_layer4[i]);

}

fclose(biases_layer4_ptr);

// Other parameters

int filtersize4 = 9; //3X3

int patchsize4 = 3;

int padsize4 = (patchsize4 - 1) / 2;
```

```
int num_filters4 = 12;

int num_channels4 = 12;

double prelu_coeff_layer4 = 0.2476;

// Convolution

double *img_fltr_4 = (double *)calloc(rows * cols * num_filters4 , sizeof(double));

double *kernel4 = (double *)malloc(filtersize4*sizeof(double));

double *img_fltr_p4 = img_fltr_4; // Pointer to img_fltr4

// double *img_fltr_4_tmp = (double *)malloc(rows * cols * sizeof(double));


img_fltr_p3 = img_fltr_3;

#pragma omp parallel for

for (int i = 0; i < num_filters4; i++)

{

  double img_fltr_4_tmp[rows * cols];

  for (int j = 0; j < num_channels4; j++)

  {

        // reading corresponding weights to kernel

        // for (int cnt_kernel = 0; cnt_kernel < filtersize4; cnt_kernel++)

        // {

        //      *(kernel4 + cnt_kernel) = weights_layer4[cnt_weight + cnt_kernel];

        // }


        imfilter(img_fltr_p3+j*cols*rows, weights_layer4+(i*num_channels4+j)*filtersize4, img_fltr_4_tmp, rows, cols, padsize4);

        imadd(img_fltr_p4+i*cols*rows, img_fltr_4_tmp, cols, rows);
```

```
                // cnt_weight = cnt_weight + filtersize4;
                // img_fltr_p3 = img_fltr_p3 + rows*cols;
            }
            PReLU(img_fltr_p4+i*cols*rows, rows, cols, biases_layer4[i], prelu_coeff_layer4);
            // img_fltr_p4 = img_fltr_p4 + rows*cols;
        }

        free(img_fltr_3);
        img_fltr_3 = NULL;
        free(kernel3);
        kernel3 = NULL;

        /////////// Layer5
        // Reading weights of 5th layer
        FILE *weights_layer5_ptr;
        weights_layer5_ptr = fopen("weights_layer5.txt", "r");
        if (weights_layer5_ptr == NULL) { printf("Error in the reading weights of 5th layer\n"); };
        double weights_layer5[1296];
        for (int i = 0; i < 1296; i++)
        {
          fscanf(weights_layer5_ptr, "%lf", &weights_layer5[i]);
        }
        fclose(weights_layer5_ptr);
        // Reading biases of 5th layer
        FILE *biases_layer5_ptr;
```

```
biases_layer5_ptr = fopen("biasess_layer5.txt", "r");

if (biases_layer5_ptr == NULL) { printf("Error in the reading biases of 5th layer\n"); };

double biases_layer5[12];

for (int i = 0; i < 12; i++)

{

  fscanf(biases_layer5_ptr, "%lf", &biases_layer5[i]);

}

fclose(biases_layer5_ptr);

// Other parameters

int filtersize5 = 9; //3X3

int patchsize5 = 3;

int padsize5 = (patchsize5 - 1) / 2;

int num_filters5 = 12;

int num_channels5 = 12;

double prelu_coeff_layer5 = 0.3495;

// Convolution

double *img_fltr_5 = (double *)calloc(rows * cols * num_filters5 , sizeof(double));

double *kernel5 = (double *)malloc(filtersize5*sizeof(double));

double *img_fltr_p5 = img_fltr_5; // Pointer to img_fltr5

// double *img_fltr_5_tmp = (double *)malloc(rows * cols * sizeof(double));



img_fltr_p4 = img_fltr_4;

#pragma omp parallel for

for (int i = 0; i < num_filters5; i++)

{
```

```
        double img_fltr_5_tmp[rows*cols];
        for (int j = 0; j < num_channels5; j++)
        {
                // reading corresponding weights to kernel
                // for (int cnt_kernel = 0; cnt_kernel < filtersize5; cnt_kernel++)
                // {
                //      *(kernel5 + cnt_kernel) = weights_layer5[cnt_weight + cnt_kernel];
                // }

                imfilter(img_fltr_p4+j*cols*rows, weights_layer5+(i*num_channels5+j)*filtersize5, img_fltr_5_tmp, rows, cols, padsize5);
                imadd(img_fltr_p5+i*cols*rows, img_fltr_5_tmp, cols, rows);

                // cnt_weight = cnt_weight + filtersize5;
                // img_fltr_p4 = img_fltr_p4 + rows*cols;
        }
        // bias_tmp = biases_layer5[i];
        PReLU(img_fltr_p5+i*cols*rows, rows, cols, biases_layer5[i], prelu_coeff_layer5);
        // img_fltr_p5 = img_fltr_p5 + rows*cols;
    }

    free(img_fltr_4);
    img_fltr_4 = NULL;
    free(kernel4);
    kernel4 = NULL;
```

```
////////// Layer6

// Reading weights of 6th layer

FILE *weights_layer6_ptr;

weights_layer6_ptr = fopen("weights_layer6.txt", "r");

if (weights_layer6_ptr == NULL) { printf("Error in the reading weights of 6th layer\n"); };

double weights_layer6[1296];

for (int i = 0; i < 1296; i++)

{

  fscanf(weights_layer6_ptr, "%lf", &weights_layer6[i]);

}

fclose(weights_layer6_ptr);

// Reading biases of 6th layer

FILE *biases_layer6_ptr;

biases_layer6_ptr = fopen("biasess_layer6.txt", "r");

if (biases_layer6_ptr == NULL) { printf("Error in the reading biases of 6th layer\n"); };

double biases_layer6[12];

for (int i = 0; i < 12; i++)

{

  fscanf(biases_layer6_ptr, "%lf", &biases_layer6[i]);

}

fclose(biases_layer6_ptr);

// Other parameters

int filtersize6 = 9; //3X3

int patchsize6 = 3;

int padsize6 = (patchsize6 - 1) / 2;
```

```
int num_filters6 = 12;

int num_channels6 = 12;

double prelu_coeff_layer6 = 0.7806;

// Convolution

double *img_fltr_6 = (double *)calloc(rows * cols * num_filters6 , sizeof(double));

double *kernel6 = (double *)malloc(filtersize6*sizeof(double));

double *img_fltr_p6 = img_fltr_6; // Pointer to img_fltr6

// double *img_fltr_6_tmp = (double *)malloc(rows * cols * sizeof(double));


img_fltr_p5 = img_fltr_5;

#pragma omp parallel for

for (int i = 0; i < num_filters6; i++)

{

  double img_fltr_6_tmp[rows*cols];

  for (int j = 0; j < num_channels6; j++)

  {

        // reading corresponding weights to kernel

        // for (int cnt_kernel = 0; cnt_kernel < filtersize6; cnt_kernel++)

        // {

        //      *(kernel6 + cnt_kernel) = weights_layer6[cnt_weight + cnt_kernel];

        // }

        imfilter(img_fltr_p5+j*cols*rows, weights_layer6+(i*num_channels6+j)*filtersize6, img_fltr_6_tmp, rows, cols, padsize6);

        imadd(img_fltr_p6+i*cols*rows, img_fltr_6_tmp, cols, rows);

        // cnt_weight = cnt_weight + filtersize6;
```

```
                // img_fltr_p5 = img_fltr_p5 + rows*cols;
            }
            // bias_tmp = biases_layer6[i];
            PReLU(img_fltr_p6+i*cols*rows, rows, cols, biases_layer6[i], prelu_coeff_layer6);
            // img_fltr_p6 = img_fltr_p6 + rows*cols;
        }

        free(img_fltr_5);
        img_fltr_5 = NULL;

        free(kernel5);
        kernel5 = NULL;

        /////////// Layer7
        // Reading weights of 7th layer
        FILE *weights_layer7_ptr;
        weights_layer7_ptr = fopen("weights_layer7.txt", "r");
        if (weights_layer7_ptr == NULL) { printf("Error in the reading weights of 7th layer\n"); };
        double weights_layer7[672];
        for (int i = 0; i < 672; i++)
        {
          fscanf(weights_layer7_ptr, "%lf", &weights_layer7[i]);
        }
        fclose(weights_layer7_ptr);
        // Reading biases of 7th layer
        FILE *biases_layer7_ptr;
```

```
biases_layer7_ptr = fopen("biasess_layer7.txt", "r");

if (biases_layer7_ptr == NULL) { printf("Error in the reading biases of 7th layer\n"); };

double biases_layer7[56];

for (int i = 0; i < 56; i++)

{

  fscanf(biases_layer7_ptr, "%lf", &biases_layer7[i]);

}

fclose(biases_layer7_ptr);

// Other parameters

int filtersize7 = 1; //1X1

int patchsize7 = 1;

int padsize7 = (patchsize7 - 1) / 2;

int num_filters7 = 56;

int num_channels7 = 12;

double prelu_coeff_layer7 = 0.0087;

// Convolution

double *img_fltr_7 = (double *)calloc(rows * cols * num_filters7 , sizeof(double));

double *kernel7 = (double *)malloc(filtersize7*sizeof(double));

double *img_fltr_p7 = img_fltr_7; // Pointer to img_fltr7

// double *img_fltr_7_tmp = (double *)malloc(rows * cols * sizeof(double));

img_fltr_p6 = img_fltr_6;

#pragma omp parallel for

for (int i = 0; i < num_filters7; i++)

{

  double img_fltr_7_tmp[rows*cols];
```

```
        for (int j = 0; j < num_channels7; j++)
        {
                // reading corresponding weights to kernel
                // for (int cnt_kernel = 0; cnt_kernel < filtersize7; cnt_kernel++)
                // {
                //      *(kernel7 + cnt_kernel) = weights_layer7[cnt_weight + cnt_kernel];
                // }

                imfilter(img_fltr_p6+j*cols*rows, weights_layer7+(i*num_channels7+j)*filtersize7, img_fltr_7_tmp, rows, cols, padsize7);
                imadd(img_fltr_p7+i*cols*rows, img_fltr_7_tmp, cols, rows);

                // cnt_weight = cnt_weight + filtersize7;

                // img_fltr_p6 = img_fltr_p6 + rows*cols;
        }
        // bias_tmp = biases_layer7[i];
        PReLU(img_fltr_p7+i*cols*rows, rows, cols, biases_layer7[i], prelu_coeff_layer7);
        // img_fltr_p7 = img_fltr_p7 + rows*cols;
      }

      free(img_fltr_6);
      img_fltr_6 = NULL;
      free(kernel6);
      kernel6 = NULL;
```

```
////////// Convolution3 ---------- Layer 8


////////// Layer8

// Reading weights of 8th layer

FILE *weights_layer8_ptr;

weights_layer8_ptr = fopen("weights_layer8.txt", "r");

if (weights_layer8_ptr == NULL) { printf("Error in the reading weights of 8th layer\n"); };

double weights_layer8[4536];

for (int i = 0; i < 4536; i++)

{

  fscanf(weights_layer8_ptr, "%lf", &weights_layer8[i]);

}

fclose(weights_layer8_ptr);

// Reading biases of 8th layer

double biases_layer8 = - 0.03262640000;

// Other parameters

int filtersize8 = 81; //9x9

int patchsize8 = 9;

int num_filters8 = 1;

int num_channels8 = 56;


// Decvolution ==> output is the img_hr

double *img_fltr_8 = (double *)calloc((rows*scale) *(cols*scale) * num_filters8 , sizeof(double));

double *kernel8 = (double *)malloc(filtersize8*sizeof(double));

double *img_fltr_8_tmp = (double *)malloc((rows*scale) *(cols*scale) * sizeof(double));
```

```
img_fltr_p7 = img_fltr_7;

// private(cnt_weight,img_fltr_p7)


for (int j = 0; j < num_channels8; j++)

{

  deconv(img_fltr_p7+j*cols*rows,                    img_fltr_8_tmp,
weights_layer8+j*filtersize8, cols, rows, scale);

  imadd(img_fltr_8, img_fltr_8_tmp, cols*scale, rows*scale);

}




for (int i=0;i<rows*scale;i++)

for (int j = 0;j<cols*scale; j++)

{

  int cnt_fnl = i*cols*scale + j;

  *(img_hr + cnt_fnl) = *(img_fltr_8 + cnt_fnl) + biases_layer8;

}


/*for ( int i = 0; i < 10; i++)

{

  printf("%f\n", *(img_hr + i));

}*/


free(img_fltr_7);

img_fltr_7 = NULL;

free(kernel7);
```

```
	kernel7 = NULL;


	free(img_fltr_8);

	img_fltr_8 = NULL;

	free(img_fltr_8_tmp);

	img_fltr_8_tmp = NULL;

	free(kernel8);

	kernel8 = NULL;


}



void imfilter(double *img, double *kernel, double *img_fltr, int rows, int cols, int padsize)

{

	// img_pad is the pointer to padded image

	// kernel is the pointer to the kernel which used for convolution

	// img_fltr is the pointer to the filtered image by applying convolution

	int cols_pad = cols + 2 * padsize;

	int rows_pad = rows + 2 * padsize;

	int i, j, cnt, cnt_pad, cnt_krnl, k1, k2;

	double sum;


	double *img_pad = (double *)malloc(rows_pad * cols_pad * sizeof(double));

	pad_image(img, img_pad, rows, cols, padsize);


	for (i = padsize; i < rows_pad - padsize; i++)
```

```
        for (j = padsize; j < cols_pad - padsize; j++)
        {
          cnt = (i - padsize)*cols + (j - padsize); // counter which shows current pixel in filtered image (central pixel in convolution window)
          sum = 0;
          cnt_krnl = 0; // counter which determines kernel elements
          for (k1 = -padsize; k1 <= padsize; k1++)
          for (k2 = -padsize; k2 <= padsize; k2++)
          {
                cnt_pad = (i + k1)*cols_pad + j + k2; // counter which shows each neighbouring pixel of padded image used for convolution with kernel
                sum = sum + (*(img_pad + cnt_pad))*(*(kernel + cnt_krnl));
                cnt_krnl++;
          }
          *(img_fltr + cnt) = sum;
        }

        free(img_pad);
        img_pad = NULL;
}

// Replicate image padding by the factor of "padsize"
void pad_image(double *img, double *img_pad, int rows, int cols, int padsize)
{ // This function receives an image and paddes its border in a replicative manner
        int cols_pad = cols + 2 * padsize;
        int rows_pad = rows + 2 * padsize;
        int i, j, k, cnt, cnt_pad, k1, k2;
        double x;
```

```
// Centeral pixels


for (i = padsize; i < rows_pad - padsize; i++)
for (j = padsize; j < cols_pad - padsize; j++)
{
  cnt_pad = i * cols_pad + j;
  cnt = (i - padsize)*(cols) + j - padsize;
  x = *(img + cnt);
  *(img_pad + cnt_pad) = x;
}
// Top and Bottom Rows
//#pragma omp parallel for private(k, cnt_pad, cnt) shared(j)
for (j = padsize; j < cols_pad - padsize; j++)
for (k = 0; k < padsize; k++)
{
  // Top Rows
  cnt_pad = j + k*cols_pad;
  cnt = j - padsize;
  *(img_pad + cnt_pad) = *(img + cnt);
  // Bottom Rows
  cnt_pad = j + (rows_pad - 1 - k)* cols_pad;
  cnt = (j - padsize) + (rows - 1)*cols;
  *(img_pad + cnt_pad) = *(img + cnt);
}
// Left and Right Columns
//#pragma omp parallel for private(k, cnt_pad, cnt) shared(i)
for (i = padsize; i < rows_pad - padsize; i++)
```

```
for (k = 0; k < padsize; k++)
{
  // Left Columns
  cnt = (i - padsize)*cols;
  cnt_pad = i*cols_pad + k;
  *(img_pad + cnt_pad) = *(img + cnt);
  // Right Columns
  cnt = (i - padsize)*cols + cols - 1;
  cnt_pad = i*cols_pad + cols_pad - 1 - k;
  *(img_pad + cnt_pad) = *(img + cnt);
}
// Corner Pixels
//#pragma omp parallel for private(k2, cnt_pad) shared(k1)
for (k1 = 0; k1 < padsize; k1++)
for (k2 = 0; k2 < padsize; k2++)
{
  // Upper Left Corner
  cnt_pad = k1*cols_pad + k2;
  *(img_pad + cnt_pad) = *(img);
  // Upper Right Corner
  cnt_pad = k1*cols_pad + cols_pad - 1 - k2;
  *(img_pad + cnt_pad) = *(img + cols - 1);
  // Lower Left Corner
  cnt_pad = (rows_pad - 1 - k1)*cols_pad + k2;
  *(img_pad + cnt_pad) = *(img + (rows - 1)*cols);
  // Lower Right Corner
  cnt_pad = (rows_pad - 1 - k1)*cols_pad + cols_pad - 1 - k2;
```

```
            *(img_pad + cnt_pad) = *(img + (rows - 1)*cols + cols - 1);
        }
}

void PReLU(double *img_fltr,int rows, int cols, double bias, double prelu_coeff)
{
        int cnt;
        #pragma omp parallel for simd collapse(2) private(cnt)
        for (int i = 0; i < rows;i++)
        for (int j = 0; j < cols; j++)
        {
          cnt = i*cols + j;
          *(img_fltr + cnt) = Max(*(img_fltr + cnt) + bias, 0) + prelu_coeff *
Min(*(img_fltr + cnt) + bias, 0);
        }
}

double Max(double a, double b)
{
        double c;
        c = a > b ? a : b;
        return c;
}

double Min(double a, double b)
{
        double c;
```

```
        c = a > b ? b : a;
        return c;
}

void imadd(double *img_fltr_sum, double *img_fltr_crnt, int cols, int rows)
{
        // *img_fltr_crnt ==> pointer to current feature map
        // *img_fltr_sum ==> pointer to the cumulutive feature map
        int cnt = 0;
        #pragma omp parallel for simd collapse(2) private(cnt)
        for (int i = 0; i < rows;i++)
        for (int j = 0; j < cols; j++)
        {
          cnt = i*cols + j;
          *(img_fltr_sum + cnt) = *(img_fltr_sum + cnt) + *(img_fltr_crnt + cnt);
        }
}

void deconv(double *img_input, double *img_output, double *kernel, int cols, int
rows, int stride)
{
        int border = 1;
        int fsize = 9;
        int rows_pad = rows + 2 * border;
        int cols_pad = cols + 2 * border;
        double *img_input_padded = (double *)malloc(rows_pad * cols_pad *
sizeof(double));
```

```
pad_image(img_input, img_input_padded, rows, cols, border);


int rows_out_pad = rows_pad * stride;

int cols_out_pad = cols_pad * stride;

double *img_output_tmp = (double *)calloc((rows_out_pad + fsize - 1)* (cols_out_pad + fsize - 1), sizeof(double));

double *kernel_modif = (double *)malloc(fsize * fsize * sizeof(double));


int idx, idy,cnt_img,cnt_img_output;

int cnt_kernel =0;


for (int i = 0; i < rows_pad; i++)

{

  for (int j = 0; j < cols_pad; j++)

  {

        cnt_img = i * cols_pad + j;

        idx = i * stride;

        idy = j * stride;

        cnt_img_output = idx * (cols_out_pad + fsize - 1) + idy; // (idx,idy) coordinate in temporal output image

        cnt_kernel = 0;


        for (int k_r = 0; k_r < fsize; k_r++)

        {

              for (int k_c = 0; k_c < fsize; k_c++)

              {

                    cnt_kernel = k_r * fsize + k_c;
```

```
                                *(img_output_tmp + cnt_img_output + k_c) =
*(img_output_tmp + cnt_img_output + k_c) + *((kernel + cnt_kernel)) *
(*(img_input_padded + cnt_img));

                        }


                        cnt_img_output = cnt_img_output + (cols_out_pad + fsize -
1);

                }

          }

        }

        int rows_out = rows*stride;

        int cols_out = cols*stride;


        for (int i = 0; i < rows_out; i++)

        for (int j = 0; j < cols_out; j++)

        {

          int i_tmp = i + ((fsize + 1) / 2) + stride*border - 1;

          int j_tmp = j + ((fsize + 1) / 2) + stride*border - 1;

          int cnt_img_out = i*cols_out + j;

          int cnt_img_out_tmp = i_tmp*(cols_out_pad + fsize - 1) + j_tmp; //
(cols-pad+fsize-1) is the number of columns in the img_out_tmp

          *(img_output + cnt_img_out) = *(img_output_tmp + cnt_img_out_tmp);



        }


        free(img_input_padded); img_input_padded = NULL;

        free(img_output_tmp); img_output_tmp = NULL;

        free(kernel_modif); kernel_modif = NULL;

}
```

```
void double_2_uint8(double *double_img, unsigned char *uint8_img, int cols, int
rows)

{

        int i, j, cnt, k;

        for (i = 0; i < rows;i++)

        for (j = 0; j < cols; j++)

        {

          cnt = i*cols + j;


          if (*(double_img + cnt) < 0)

                  * (uint8_img + cnt) = 0;

          if (*(double_img + cnt) > 255)

                  * (uint8_img + cnt) = 255;


          for (k = 0; k < 255; k++)

          {

                  if (*(double_img + cnt) >= k && *(double_img + cnt) < (k+0.5))

                  *(uint8_img + cnt) =  k;


                  if (*(double_img + cnt) >= (k+0.5) && *(double_img + cnt) <
(k+1))

                          *(uint8_img + cnt) = k + 1;

          }


        }

}
```