Accelerating Discrete Cosine Transform on FPGA.

Y. Kirthi Vignan 2020122004
E. Siddharth Pavan 2018102033

Introduction

DCT(Discrete Cosine Transform) is a type of transform, where we process the input using weighted cosine transform of different frequencies.

It is a widely used transformation technique in the domain of signal processing and data compression.

Here we are implementing a 2D-DCT which is very much used in the JPEG compression of a image.

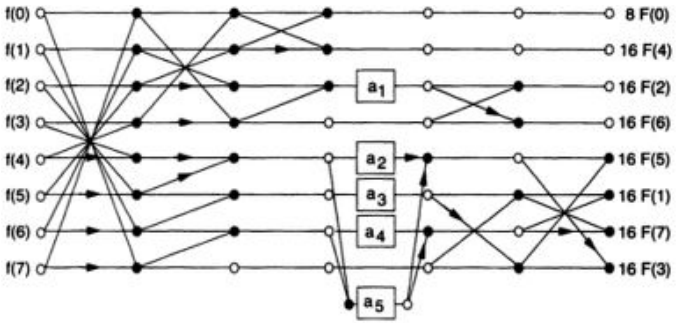
The output provided by the transform will be considered as the coefficients corresponding to the frequency associated to the image thereby utilising less space by eliminating unnecessary terms.

1D-DCT

A 1D-DCT is used in the analysis of signal processing and transformation.

This is a very basic implementation of the cosine transform where the approach is often realised with a “Butterfly diagram”.

Here the x_n = is the input signal and X_k = is the DCT coefficient corresponding to the frequency.



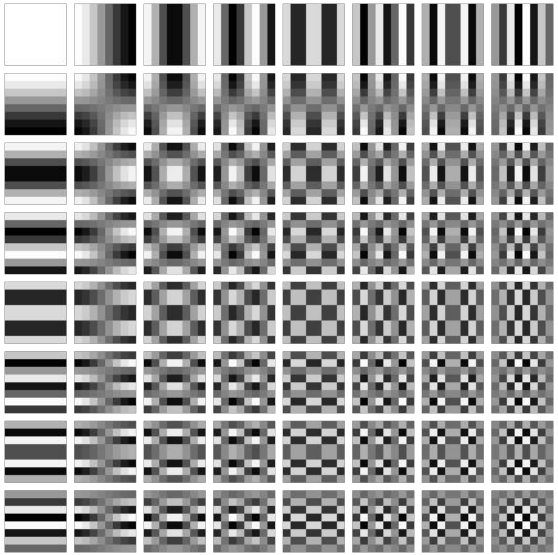
$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \qquad \text{for } k = 0, \ldots N-1 .$$

2D-DCT implementation

A 2D DCT is intuitively an extended variant of a classical DCT transform, where the frequancy change occur in both the co-ordinate axes.

This is best used for the applications involving image or video processing/ compression.

There are multi-dimensional implementation of the transform which are used for bigger applications.



DCT operation

$$\mathrm{DCT}(i,j) = \frac{1}{\sqrt{2N}} C(i)\, C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \mathrm{pixel}(x,y) \cos\left[\frac{(2x+1)i\pi}{2N}\right] \cos\left[\frac{(2y+1)j\pi}{2N}\right]$$

$$C(x) = \frac{1}{\sqrt{2}} \text{ if } x \text{ is } 0, \text{ else } 1 \text{ if } x > 0$$

Image Processing

Here, the image is broken down into 8x8 chunks. Each of these matrices(chunks) are then cosine transformed in the way described previously.

Then we have a Coefficient matrix as output. This matrix describes about the 8x8 image based on its frequency.

Then we quantise the coefficient matrix based on the required image quality. Here higher frequency terms are neglected and lower frequency components are stored.

Hence the name

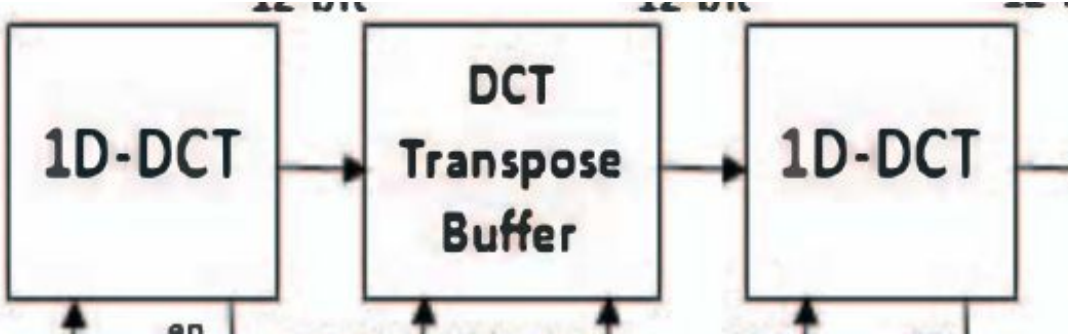
Design implementation

The very naive approach to perform the dct on the image chunk is to multiply and accumulate with each type of the frequency components across both the axes.

This would lead to more processing time for each block.

Hence a modification is brought up to the actual approach wherein we multiply each 1-DCT matrix and its transpose to the 8x8 image chunk to get its corresponding coefficients.

This approach would help us to compute coefficients faster and parallelising is also possible here.



$$DCT = \begin{bmatrix} c_{00} & c_{01} & \dots & c_{07} \\ c_{10} & c_{11} & \dots & c_{17} \\ \vdots & \vdots & & \vdots \\ c_{70} & c_{71} & \dots & c_{77} \end{bmatrix} \begin{bmatrix} X_{00} & X_{01} & \dots & X_{07} \\ X_{10} & X_{11} & \dots & X_{17} \\ \vdots & \vdots & & \vdots \\ X_{70} & X_{71} & \dots & X_{77} \end{bmatrix} \begin{bmatrix} c_{00} & c_{10} & \dots & c_{70} \\ c_{01} & c_{11} & \dots & c_{71} \\ \vdots & \vdots & & \vdots \\ c_{07} & c_{17} & \dots & c_{77} \end{bmatrix}$$

The transformation matrix used in kernel

```
float matA[64]={
    g,  a,  b,  c,  g,  d,  e,  f,
    g,  c,  e, -f, -g, -a, -b, -d,
    g,  d, -e, -a, -g,  f,  b,  c,
    g,  f, -b, -d,  g,  c, -e, -a,
    g, -f, -b,  d,  g, -c, -e,  a,
    g, -d, -e,  a, -g, -f,  b, -c,
    g, -c,  e,  f, -g,  a, -b,  d,
    g, -a,  b, -c,  g, -d,  e,  f
};

float matAt[64]={
        g,g,g,g,g,g,g,g,
        a,c,d,f,-f,-d,-c,-a,
        b,e,-e,-b,-b,-e,e,b,
        c,-f,-a,-d,d,a,f,-c,
        g,-g,-g,g,g,-g,-g,g,
        d,-a,f,c,-c,-f,a,-d,
        e,-b,b,-e,-e,b,-b,e,
        f,-d,c,-a,a,-c,d,f
    };
```

```
const float a = 0.4903926402;
const float b = 0.46193976625;
const float c = 0.41573480615;
const float d = 0.27778511651;
const float e = 0.19134171618;
const float f= 0.097545161;
const float g = 0.35355339059;
```

```
    read1: for (unsigned int j = 0; j < chunk_size; j++) {
        #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
        v1_buffer[j] = in1[i + j];
    }

for (int r = 0; r < 8; r++) {
    for (int c = 0; c < 8; c++) {
        float sum = 0.0;
        for (int k = 0; k < 8; k++)
            sum = sum + matA[r * 8 + k] * v1_buffer[k * 8 + c];
        matC[r * 8 + c] = sum;

    }
}
for (int r = 0; r < 8; r++) {
    for (int c = 0; c < 8; c++) {
        float sum = 0.0;
        for (int k = 0; k < 8; k++)
            sum = sum + matC[r * 8 + k] * matAt[k * 8 + c];
        matD[r * 8 + c] = sum;

    }
}

    //Burst reading B and calculating C and Burst writing
    // to  Global memory
    vadd_writeC: for (unsigned int j = 0; j < chunk_size; j++) {
        #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
        //perform vector addition
        out_r[i+j] = matD[j];
    }
```

Algorithm for DCT

Roofline Analysis

No.of DSPs =6800
Each DSP can do 1 MAC operation
The Clock Frequency of FPGA is 250MHz

The image taken for DFT is HD image so its dimensions are 1280*720=9,21,600 pixels
In our algorithm we take 8x8 pixels and apply 2d DFT transform that is 2 matrix multiplications with 2 8x8 matrices.

In Matrix multiplication generation of each element requires 8MAC operations .So total 8x8 matrix multiplication takes about 64x8 MAC operations =512 MAC operations for 64 pixels so it takes about =73,72,800 MAC operations in total for 1 matrix operation for 2 we need 1,47,45,600MAC operations

The FPGA can perform 6800*250 MAC operations =17,00,000/second

The 921600 pixels require 1280*720 bytes to transfer =0.99216 Megabyte The input bandwidth is 16GB/sec

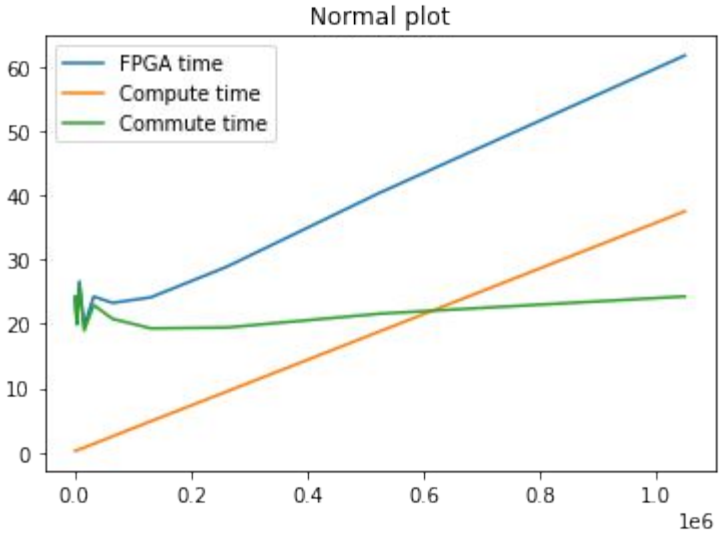
By this we can conclude that it is compute bound

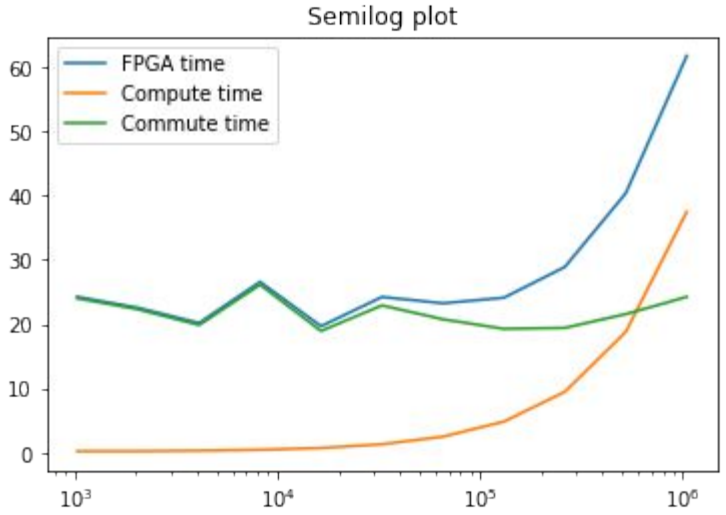
Results

Hardware Emulation Result

Vector size (image taken as 1 d)	Computation time
2048	58014.2ms
2048*2	65016.1ms
2048*4	66015.3ms

Hardware Result





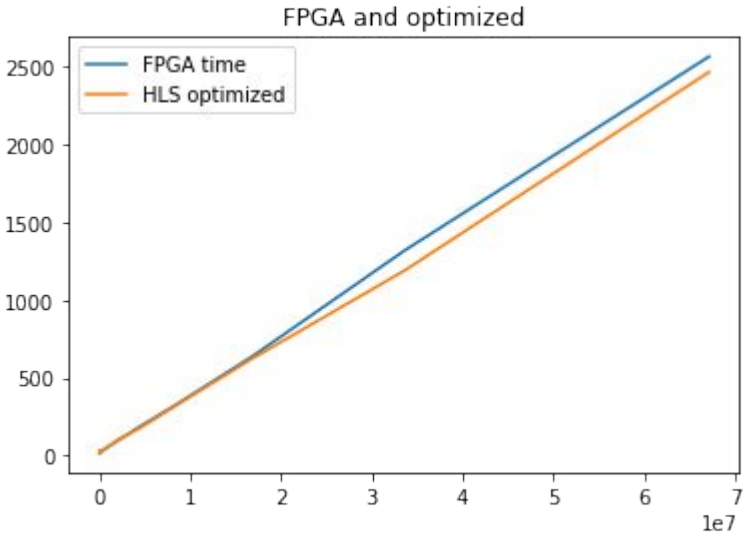
We found that the communication time is relatively constant and the compute time varies.

Result

We found that for HD image of 1280*720 we get 32.93ms as compute time and 23.783ms as communication time

For 4K image of 3840 x 2160 we get 295.057ms and compute time as 32.81 ms

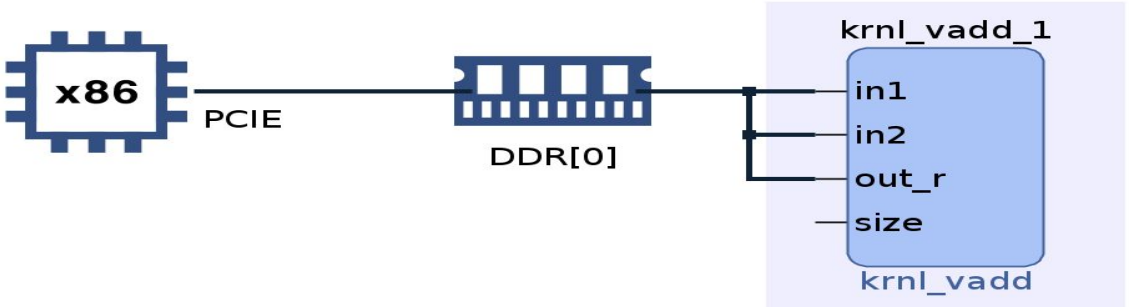
With HLS unroll and HLS loop_flatten




```
    //Per iteration of this loop perform BUFFER_SIZE vector addition
    for (unsigned int i = 0; i < size; i += BUFFER_SIZE) {
       #pragma HLS LOOP_TRIPCOUNT min=c_len max=c_len
        unsigned int chunk_size = BUFFER_SIZE;
        //boundary checks
        if ((i + BUFFER_SIZE) > size)
            chunk_size = size - i;

        read1: for (unsigned int j = 0; j < chunk_size; j++) {

           #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
            v1_buffer[j] = in1[i + j];
        }

    for (int r = 0; r < 8; r++) {
        for (int c = 0; c < 8; c++) {
#pragma HLS loop_flatten
            float sum = 0.0;
            for (int k = 0; k < 8; k++)
#pragma HLS unroll
                sum = sum + matA[r * 8 + k] * v1_buffer[k * 8 + c];
            matC[r * 8 + c] = sum;

        }
    }
    for (int r = 0; r < 8; r++) {

        for (int c = 0; c < 8; c++) {
#pragma HLS loop_flatten
            float sum = 0.0;
            for (int k = 0; k < 8; k++)
#pragma HLS unroll
                sum = sum + matC[r * 8 + k] * matAt[k * 8 + c];
            matD[r * 8 + c] = sum;
```

Comparison of Designs

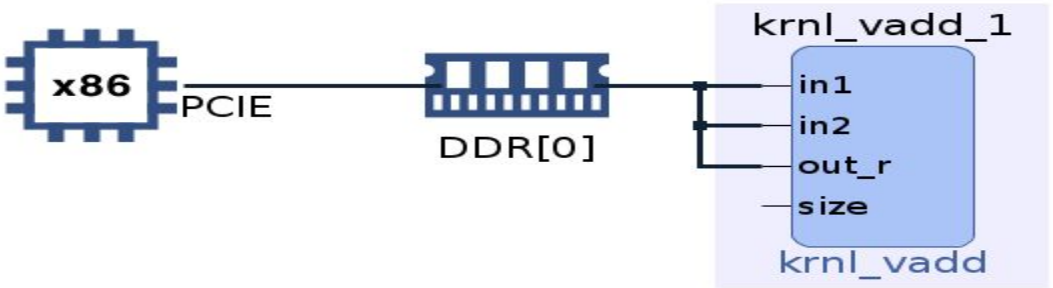
For Pragma HLS



Resources

LUT: 25,110 (2.12 %)
BRAM: 4 (0.19 %)
URAM: 0 (N/A)
Register: 31,725 (N/A)
DSP: 80 (1.17 %)

Without HLS pragma



Resources

LUT: 9,828 (0.83 %)
BRAM: 9 (0.42 %)
URAM: 0 (N/A)
Register: 12,188 (N/A)
DSP: 80 (1.17 %)

THANK YOU