High Performance Computing – A.A. 2023/24

Discrete Wavelet Transform with Lifting Scheme – Report

M. Roscica, S. Verrilli

Department of Computer Science, Machine Learning and Big Data, University of Naples Parthenope

E–mail: *martina.roscica001@studenti.uniparthenope.it, stefano.verrilli001@studenti.uniparthenope.it*

Abstract

The purpose of this report is to introduce Discrete Wavelets Transform and Cohen–Daubechies–Feauveau (CDF) 9/7 wavelet. Next we discuss the Lifting Scheme design in detail and how this algorithm can be used to generate bi-orthogonal wavelets such as CDF in order to compress images with integer-only transformations. Last section is dedicated to algorithm and implementation details of sequential and parallel versions.

Key words. Wavelets – CDF 9/7 – Lifting – Parallel computing – CUDA – Signal processing

1. INTRODUCTION

Anilkumar and Beulet (2015), Sweldens (1996), Sweldens and Schröder (2005), Uytterhoeven et al. (1997)The following report section is meant to provide a brief introduction to wavelet transform and its advantages compared to other transformations, as Fourier transform. The WT emerges as a crucial tool for analyzing non-stationary signals, proposing a solution with versatile applications such as signal analysis, denoising, and data compression. Its advantages lie in its ability to localize information both *in the time and frequency domains* as in fig.1, making it particularly suitable for processing signals with varying features over specific time and frequency intervals, and it leads to *sparse representation* of data.

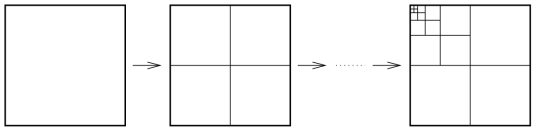

Fig. 1: Two-dimensional wavelet transform.

An innovative approach to design discrete wavelet transforms is the *Lifting Scheme*, introduced in 1996 by Sweldens, offering several benefits in terms of computational complexity reduction, hardware design simplified, cheaper inverse transform compared to traditional methods. The generic scheme of the algorithm is shown in fig.2.

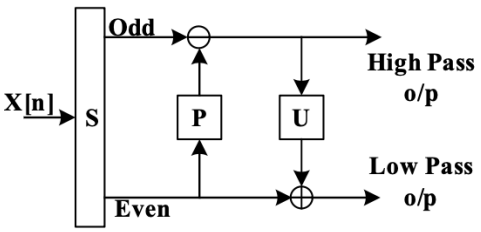

Fig. 2: Lifting scheme for DWT.

The flow of a wavelet transform algorithm that uses the lifting scheme consists of three steps:

- **Split**, dividing the input signal into odd γ_k and even λ_k components.

$$\gamma[k] \leftarrow x[2k]$$

$$\lambda[k] \leftarrow x[2k+1]$$

Taps	Low Pass Filter	Taps	High Pass Filter
0, 8	0.026748757410	0, 6	0.091271763114
1, 7	–0.016864118442	1, 5	–0.057543526228
2, 6	–0.078223266528	2, 4	–0.591271763114
3, 5	0.266864118442	3	1.111508705245
4	0.602949018326		

Fig. 3: Daubechies 9/7 filters coefficients.

- **Dual lifting** – **Prediction** of the odd samples from the even samples using a prediction filter P, an interpolation function. This step helps to estimate the values of odd samples based on the neighboring even samples.

$$\gamma[k] = \gamma[k] - P(\lambda[k])$$

- **Primal lifting** – **Update** the even samples using the predicted odd samples and the original even samples. This step refines the even samples based on the prediction made in the previous step:

$$\lambda[k] = \lambda[k] - U(\gamma[k])$$

Eventually it can be added a last step of normalization.
The inverse process follows the same steps backward to return back to *quo-ante* status.
From this point, our focal point will be the CDF 9/7, a specific type of wavelet transform within the CDF family wavelet, known as Cohen-Daubechies-Feauveau. They are widely used for their bi-orthogonal property, since CDFs have a set of filters for analysis and another set for synthesis and they are respectively orthogonal to each other so, for compression tasks, this can be translated in terms of transformations without significant distortions. CDF 9/7 means this wavelet has 9 coefficients for analysis filter and 7 for synthesis filter and their canonical values are shown in fig.3. The lifting scheme, applied to CDF 9/7 transform, contains 5 fundamenal steps shown in fig.4:

- **Predict I**

$$I_{2i+1} = X_{j,2i+1} + \alpha(X_{j,2i} + j, X_{2i+2})$$

- **Update I**

$$I_{2i} = X_{j,2i} + \beta(X_{j,2i-1} + j, X_{2i+1})$$

- **Predict II**

$$P_{j-1,i} = I_{2i-1} + \gamma(I_{j,2i-2} + I_{2i})$$

- **Update II**

$$U_{j-1,i} = I_{2i} + \delta(P_{j-1,i} + P_{j-1,i+1})$$

- **Normalization**

$$H_{j-1,i} = K(P_{j-1,i}) \quad G_{j-1,i} = \frac{1}{K}(U_{j-1,i})$$

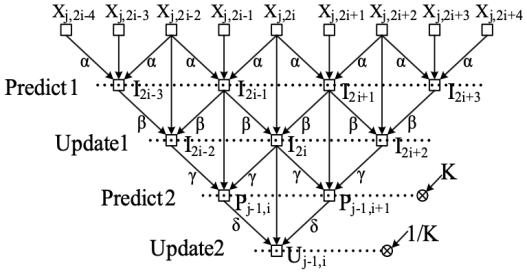

Fig. 4: Lifting scheme for CDF 9/7.

Following the default approach, coefficients values are defined in fig.5.

Coefficients	Floating Point Value	Coefficients	Floating Point Value
α	–1.586134342	β	–0.052980118
γ	0.882911075	δ	0.443506852
–K	–1.230174105	1/K	0.812893066

Fig. 5: Coefficients for CDF 9/7.

2. ALGORITHM DESCRIPTION

In this report, our primary focus is to demonstrate one of the various tasks facilitated by CDF 9/7 wavelets and their implementation: compression.
As mentioned earlier, CDF is extensively utilized for compression, much like JPEG 2000. Here, we present an implementation for compressing gray-scale images with lossless compression using CDF 9/7, capable of performing both lossless and lossy compression.

2.1. Sequential implementation

Following algorithm guidelines in the section 1, we establish a number of levels equal to 3. To take a glance to the decomposition see fig.6 below. We proceed to load the image and dynamically allocate a float matrix to access the pixels, then storing image dimensions in variable.

- Compression:
  Utilizing the "forwardDWT97()" function, we conduct both row and column decomposition with "lifting97()" for each specified number of levels.By defining the constant values $\alpha, \beta, \gamma, \delta, K$ and IK, we execute the CDF 9/7 wavelet steps, presenting the intermediate results on the image display.(7a and 7c)

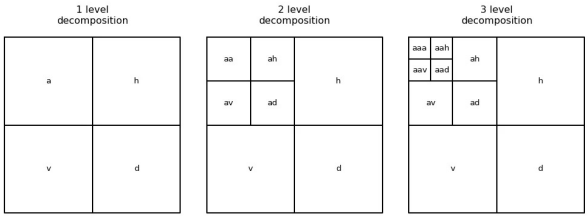

Fig. 6: Decomposition for 3 levels.

- Standard techniques:
  Techniques involved in this step are quantization, thresholding and run length encoding since images are composed by pixels of integer values of intensity.
- Reconstruction:
  To show the reconstructed image using inverse CDF 9/7, calling the function "Ilifting97()", it reverses the previous steps, undoing Update-Prediction steps of the forward procedure. After that, we can compare the reconstructed output with the given input.(7e and 7g)

2.2. Parallel implementation

To understand how parallel approach works:

- Setup
  To organize the settings of the parallelization, are defined nThreadPerBlock of BLOCK_SIZE=16 and nBlocks, that indicates how many blocks to cover the entire image, is given by image dimension(width and height) and number of threads per block (on x and y). Assuming width=height=512, we compute width/nThreadPerBlock.x and add 1 if the dimension is not perfectly divisible by nThreadPerBlock, same process for height. Size is measured as total dimension of float image, width*height*sizeof(float).
  Device float matrices: d_image, temp.
  Host float matrices: h_image.
- The dwt97Rows()
  executes the Discrete Wavelet Transform (DWT) 9/7 on the rows of the image. This kernel is launched with multiple threads, each of which processes a single row of the image in parallel. Each thread operates on a portion of the image and computes the wavelet transformation on its own row of the image in order to process multiple rows simultaneously. Synchronization is implemented with __threadfence() and __syncthreads().
  Each step of Lifting Scheme is performed with a prior boundary check to ensure that indices are within valid limits.(7b)
- The transposeMatrix() function
  is performed to deal with columns, after rows, through the re-use of the dwt97Rows() function, following the same strategy on the temporary matrix.(7d)
- With Idwt97Rows()
  it is performed the inverse procedure, undoing each step of the Lifting Scheme.(7f)
  Then, applying both transposeMatrix and inverse Idwt97Row the output is shown as in fig.(7h)

3. ENVIRONMENT and ROUTINES

The implementations of the algorithm in sequential and parallel are provided in the C programming language. For the parallel approach, the **CUDA** library has been utilized to leverage the potential of parallelization.

Library SDL2 has been included to handle images and execute basic operations as loading, saving and displaying thanks to *SDL_Surface*, a struct to represent a surface of rendering or an image that contains information about the dimension, number of bits, pointer to image contents.

The running machine provided by the University of Naples Parthenope (2019) for developing and testing the parallel algorithm is named purpleJeans, dedicated to research in the fields of machine learning and big data. Its specifications include 4 nodes with the following configuration:

- CPU/node: 2 Intel(R) Xeon(R) Xeon 16-Core 5218 2.3GHz 22MB CPUs
- Cores/Node: 32
- Memory/Node: 192 GB

4. TEST

To assess the capabilities and the quality of the compression of this algorithm, black-and-white images in bitmap format with dimensions of 512x512 are provided for computational simplicity. The sample below includes commonly used images in the field of image processing, such as "Barbara" and "Lena".
To compile the algorithm, follow user guide, also in section 6.

5. PERFORMANCES

Among the advantages enumerated by parallel computing, in comparison to sequential computing, time stands out as the key aspect. To demonstrate the time benefits achieved by processing the algorithm in parallel versus sequentially, the execution times of

Task	Sequential (ms)	Parallel (ms)
Direct DWT	30.0ms	0.00035ms
Inverse DWT	40.0ms	0.00046ms
Total	70.0ms	0.00081ms

Table 1: Comparison of execution time between sequential and parallel implementation.

the two implementations are reported in Table 1. Although the algorithm requires rather short processing times, given the small size of the images, the sequential approach efficiently minimizes execution times, ensuring significant improvements for data of higher dimensionality.

REFERENCES

Anilkumar, P. H. and Beulet, P. A. S. 2015, Indian Journal of science and technology, 8

Sweldens, W. 1996, ZAMM-Zeitschrift fur Angewandte Mathematik und Mechanik, 76, 41

Sweldens, W. and Schröder, P. 2005, Wavelets in the Geosciences,), 72

Uytterhoeven, G., Roose, D., and Bultheel, A. 1997, ITA-Wavelets Report WP, 1

(a) (b)

(c) (d)

(e) (f)

(g) (h)

Fig. 7: On first column, results of first and second direct step (a) and (c), then the inverse steps (e) and (g) of sequential algorithm. On second column, results of direct steps (b) and inverse steps (f) and (h) of parallel algorithm.

6.1. User guide - README.md

The compilation of this sequential implementation is completely managed by the compile.sh file that you can find in this folder. To compile the **main.c/.cu** file you need to run the following command while in this folder: `source ./compile.sh` The file should be able to successfully link the SDL2 libraries contained in the lib folder.

6.2. Guide for execution of sequential

To execute the actual sequential version of lifting, as for the parallel one, you need to specify some parameters, let's break them down:

- FilePath: First parameter for execution is the filepath of the BMP file to compress, some sample images are contained in Images folder (../Images/)
- Compression: This is a boolean parameters used to define if the image has to be compressed after the forward pass of DWT. This parameter also indicate if the last other 2 parameters have to be inserted, 1 stands for compression and 0 for no need of compresssion.
- Quantization_Step: **In case you have set the Compression parameter as 1**, this parameter needs to be included to define the step size between each quantization level.
- Threshold Value: **In case you have set the Compression parameter as 1**, this parameter is utilized to set the limit value for which each pixel below will be equal to 0 in the output image.

Once parameters have been carefully selected and defined we can run the program with one of the following forms:

- `./DWT_sequential ../Images/barbara.bmp 0` (No compression in place)
- `./DWT_sequential../Images/barbara.bmp 1 100 100` (Compression with Quantization and Thresholding)

6.3. Guide for execution of parallel

To execute the actual CUDA version of lifting you need to specify some paramentes, let's break them down:

- FilePath: First parameter for execution is the filepath of the BMP file to compress, some sample images are contained in Images folder (../Images/)
- Compression: This is a boolean parameters used to define if the image has to be compressed after the forward pass of DWT. This parameter also indicate if the last other 2 parameters have to be inserted, 1 stands for compression and 0 for no need of compresssion.
- Quantization_Step: **In case you have set the Compression parameter as 1**, this parameter needs to be included to define the step size between each quantization level.
- Threshold Value: **In case you have set the Compression parameter as 1**, this parameter is utilized to set the limit value for which each pixel below will be equal to 0 in the output image.

Once parameters have been carefully selected and defined we can run the program with one of the following forms:

- `./my_run.sh ../Images/barbara.bmp 0`
- `./my_run.sh ../Images/barbara.bmp 1 100 100`

6.4. Reviewing the Result

To visualize the results produced by the program you just need to move into the **Results** folder in which you will find the result of the forward transform and the reconstructed image (compressed or not).

Sequential Implementation – FWT_cpu.c

```
#include <stdio.h>

#define ALPHA -1.586134342
#define BETA -0.05298011854
#define GAMMA 0.8829110762
#define DELTA 0.4435068522
#define K 1.230174105
#define IK 0.812893066

void lifting97(float*, int);
void Ilifting97(float*,int);
void forwardDWT97(float*, int, int,int);
void InverseDWT97(float*, int, int ,int );



float* temp=0;

void lifting97(float* data, int size) {
    int i;

    // Predict 1
    for (i = 1; i < size-2; i += 2) {
        data[i] += ALPHA * (data[i - 1] + data[i + 1]);
    }
    data[size-1] += ALPHA*data[size-2];

    // Update 1
    for (i = 2; i < size-1; i += 2) {
        data[i] += BETA * (data[i - 1] + data[i + 1]);
    }
    data[0]+=BETA*data[1];

    // Predict 2
    for (i = 1; i < size - 2; i += 2) {
        data[i] += GAMMA * (data[i - 1] + data[i + 1]);
    }
    data[size-1] += GAMMA*data[size-2];

    // Update 2
    for (i = 2; i < size-1; i += 2) {
        data[i] += DELTA * (data[i - 1] + data[i + 1]);
    }
    data[0] += DELTA*data[1];

    for (i = 0; i < size; i++) {
        if(i%2) data[i]*=K;  // Scaling factor for the 9/7 wavelet transform
        else data[i]*=IK;
    }

  if (temp==0) temp = (float*)malloc(size*sizeof(float));
    for(i=0;i<size;i++){
        if(i%2==0) temp[i/2] = data[i];
        else temp[size/2+ i/2] = data[i];
    }
    for(i=0;i<size;i++) data[i] = temp[i];
}


void Ilifting97(float* data,int size){


    if(temp == 0) temp = (float *) malloc(size*sizeof(float));
    for(int i=0;i<size/2;i++){
        temp[i*2] = data[i];
        temp[i*2+1] = data[i+size/2];
    }
```

```
    for(int i=0;i<size;i++) data[i] = temp[i];

    //Undo scaling
    for(int i=0;i<size;i++){
        if(i%2) data[i] /= K;
        else data[i]/= IK;
    }
    //Undo update 2
    for(int i=2;i<size-1;i+=2){
        data[i]+=(-DELTA)*(data[i-1]+ data[i+1]);
    }
    data[0] += (-DELTA)*data[1];

    //Undo predict 2
    for(int i=1;i<size-2;i+=2){
        data[i]+= (-GAMMA)*(data[i-1]+data[i+1]);
    }
    data[size-1] +=(-GAMMA)*data[size-2];

    //Undo update 1
    for(int i=2;i<size-1;i+=2){
        data[i]-= BETA*(data[i-1]+data[i+1]);
    }
    data[0]+=(-BETA)*data[1];


    //Undo predict 1
    for(int i=1;i<size-2;i+=2){
        data[i] += (-ALPHA)*(data[i-1]+data[i+1]);
    }
    data[size-1] += (-ALPHA)*data[size-2];

}


// Forward 9/7 wavelet transform on a 2D image
void forwardDWT97(float* image, int width, int height,int levels) {
    int i, j;

for(int k = 0;k<levels;k++){

        //Decompose rows
         for (i = 0; i < height; i++) {
            lifting97(image + i * width,width);
        }

        // Decompose columns
        for (j = 0; j < width; j++) {
            float column[height];
            for (i = 0; i < height; i++) {
                column[i] = image[i * width + j];
            }

            lifting97(column, height);

            for (i = 0; i < height; i++) {
                image[i * width + j] = column[i];
            }
        }
}
}

void InverseDWT97(float* image, int width, int height,int levels) {
    int i, j;

for(int k=levels-1;k>=0;k--){
```

```
    // Inverse rows
        for (i = 0; i < height; i++) {
            Ilifting97(image + i * width, width);
        }


    for (j = 0; j < width; j++) {
            float column[height];
            for (i = 0; i < height; i++) {
                column[i] = image[i * width + j];
            }

            Ilifting97(column, height);

            for (i = 0; i < height; i++) {
                image[i * width + j] = column[i];
            }
        }

}
}
```

Main for CPU algorithm – main.c

```
#include <stdio.h>
#include "../Shared/utils.c"
#include "FWT_cpu.c"
#include <time.h>
#include "../Shared/compression_utils.c"

int main(int argc,const char *argv[]) {
    float QuantizationStep,Threshold;

    if (argn < 2 || argn > 5) {
        printf("Usage: %s <file_path> <NeedsCompression> [<QuantizationStep> <
            ThresholdValue>]\n", argv[0]);
        return 1;
    }

    // The filepath is stored in argv[1]
    const char* filename = argv[1];
    // The flag is stored in argv[2]
    int flag = atoi(argv[2]);

    if (flag != 0 && flag != 1) {
        printf("Invalid flag. Flag must be either 0 or 1.\n");
        return 1;
    }

    if (flag == 1 && argn != 5) {
        printf("Flag is 1, but expected additional arguments are missing.\n");
        return 1;
    }

    // If flag is 1, we expect two additional arguments
    if (flag == 1) {
        // Process additional arguments
        QuantizationStep = atoi(argv[3]);
        Threshold = atoi(argv[4]);
    }

    int width, height;

    // Load grayscale image
    unsigned char* image = loadImage(filename, &width, &height);

    if (!image) {
        return 1; // Error loading image
    }

    // Convert the image to float for the transform
    float *floatImage = (float *)malloc(width * height * sizeof(float));
    for (int i = 0; i < width * height; i++) {
        floatImage[i] = (float)image[i];
    }

    size_t original_size_bytes = get_array_size_bytes(floatImage, width*height);

    // Perform the forward CDF 9/7 wavelet transform for a specified number of levels
    int levels=1;

    clock_t StartTimeDirect, EndTimeDirect,StartTimeInverse,EndTimeInverse;
    double ExecutionTimeDirect = 0,ExecutionTimeInverse = 0;

    StartTimeDirect = clock();
    forwardDWT97(floatImage,width,height,levels);
    EndTimeDirect = clock();
    saveImage("Results/direct.bmp",floatImage,width,height);
    ExecutionTimeDirect = ((double)(EndTimeDirect - StartTimeDirect))/ (CLOCKS_PER_SEC);
    printf("execution time: %f \n",ExecutionTimeDirect*1000);
```

```
    if(flag){
        quantize_image(floatImage,width,height,10.0);
        threshold_image(floatImage,width,height,100.0);
        count_non_zero_elements(floatImage,width,height);
        CompressedSegment* compressed_data;
        int compressed_size;
        compress_image_rle(floatImage,width,height,&compressed_data,&compressed_size);

        printf("Original Size:%zu\n",sizeof(float)*width*height);
        printf("Compressed size:%zu\n",sizeof(compressed_data)*compressed_size);
        printf("Memory saved:%zu\n",(sizeof(float)*width*height) - (sizeof(
            compressed_data)*compressed_size));
    }

    StartTimeInverse = clock();
    InverseDWT97(floatImage,width,height,levels);
    EndTimeInverse = clock();

    saveImage("Results/inverse.bmp",floatImage,width,height);
    ExecutionTimeInverse = ((double) (EndTimeInverse - StartTimeInverse))/ (
        CLOCKS_PER_SEC);
    printf("execution time: %f \n",ExecutionTimeInverse*1000);

    free(floatImage);
    free(image);

    return 0;
}
```

Parallel Implementation – DWT_gpu.cu

```
#include <cuda.h>
#include <stdio.h>

#define BLOCK_SIZE 16
#define ALPHA -1.586134342
#define BETA -0.05298011854
#define GAMMA 0.8829110762
#define DELTA 0.4435068522
#define K 1.230174105
#define IK 0.812893066

struct GeneralInfo{
    int width;
    int height;
    dim3 nBlocks;
    dim3 nThreadPerBlock;
};

__global__ void dwt97Rows(float *data, int width, int height,float coeff1, float coeff2)
    ;
__global__ void Normalization(float* data,int width,int height);
__global__ void INormalization(float *data,int width,int height);
__global__ void InverseDwt97(float *data, int width, int height,float coeff1,float
    coeff2);
__global__ void ISaveData(float*input,float* temp,int width,int height);
__global__ void SaveData(float*input,float*temp, int width,int height);
__global__ void transposeMatrix(float *input, float *output, int rows, int cols);

void DWTForward97(float *h_image, int width, int height);
void DWTInverse97(float *h_image,int width,int height);
void StandardProcedure(float *data,struct GeneralInfo *general_info, bool needsTranspose
    ,bool inverse);

/*
   Normalize the values previusly elaborated by predict and update 1->2.

   This function implements the last step of lifting approach using K and IK
       normalization constants for respectively
   even and odd elements. The entire computation is conducted in place using a thread
       setting besed on blocks where
   each thread is responsable for a single element in the block

   @param data: Vectorized version of the original image expressed in float elements
       vector.
   @param width: Width of the original bidimensional matrix.
   @param height: Height of the original bidimensional matrix.
   @return None: The entire operation is conducted in place.
   @see dwt97Rows, StandardProcedure, DWTForward97
*/
__global__ void Normalization(float* data,int width,int height){
    int col_idx = blockIdx.x * blockDim.x + threadIdx.x;
    int row_idx = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = row_idx * width + col_idx;
    __syncthreads();
    // Check if within the valid range
    if (row_idx < height && col_idx < width) {

        if (col_idx % 2 == 0) {
            data[idx] *= K;
        }

        if (col_idx % 2 == 1) {
            data[idx] *= IK;
        }
    }
}
```

```
/*
    Fundamental step to perform Forward pass of lifting by implementing predict and
        update step.

    Through the parametrizatiozation of this procedure is possible to perform both
        predict1 and update1 with ALPHA
    and BETA constant and predict2 and update2 with GAMMA and DELTA parameters. The
        smoothing process is performed
    again inplace and number of thread is evenly distributed between even and odd
        elements.

    This procedure is conducted separately alongside columns and rows and the final
        result can be obtained by applying
    this operation first on rows and then, onto the result, on cols or viceversa.

    Complete prodecure:
     - DWt97Rows(Predict1, Update1)
     - DWT97Rows(Predict2, Update2) (after transpose)
     - Normalization
     - SaveData

    ------
    IMPORTANT: To ensure proper utilization of updated values a __threadfance() function
        is used between prediction
    and updating step. This function call is mandatory since we need to update the data
        vector with the new computed
    values in a sequential style, without it will result in conflicts throughout the
        image.
    ------

    @param data: Vectorized version of the original image expressed in float elements
        vector.
    @param width: Width of the original bidimensional matrix.
    @param height: Height of the original bidimensional matrix.
    @param coeff1: ALPHA/GAMMA parameter used in lifting procedure
    @param coeff2: BETA/DELTA parameter used in lifting procedure
    @return None: The entire operation is conducted in place.
    @see: InverseDw797, StandardProcedure, SaveData
*/
__global__ void dwt97Rows(float *data, int width, int height,float coeff1,float coeff2)
    {
    int col_idx = blockIdx.x * blockDim.x + threadIdx.x;
    int row_idx = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = row_idx * width + col_idx;

    __syncthreads();
    // Check if within the valid range
    if (row_idx < height && col_idx < width) {
        // Prediction Step

        if (col_idx % 2 == 1 && col_idx < (width -2)) {
                data[idx] += coeff1 * (data[(idx-1)] + data[(idx + 1)]);
            }

        if(col_idx %2 == 1 && col_idx == (width -1)){
                data[idx] += (coeff1 * (data[idx-1]));
            }

        __threadfence();
        // Update Step

        if (col_idx %2 == 0 && col_idx < (width - 1) && col_idx > 0){
                data[idx] += (coeff2 * (data[(idx-1)] + data[(idx+1)]));
        }

        if(col_idx % 2 == 0 && col_idx == 0){
                data[idx] += (coeff2 *(data[(idx+1)]));
        }
```

```
        __threadfence();}}

/*
    Fundamental step to perform Inverse pass of lifting, can be utilized to preform
        Inverse Update1/2 and Inverse Predict 1/2

    As for what happend with the forward pass, we can parametrize the function to perform
         different predict and upadte
    steps by inverting the coefficients. In fact the inverse procedure is based on the
        reproposition of the forward steps
    in reverse order to achive a final reconstruction:
    ISaveData -> INormalization -> InverseDWT97(Update2, Predict2) -> InverseDWT97(Update1
        , Predict1)

    _______
    IMPORTANT: To ensure proper utilization of updated values a __threadfance() function
        is used between prediction
    and updating step. This function call is mandatory since we need to update the data
        vector with the new computed
    values in a sequential style, without it will result in conflicts throughout the
        image.
    _______

    @param data,width,height: @see dwt97Rows for details
    @param coeff1: BETA/DELTA parameter used in lifting procedure
    @param coeff2: ALPHA/GAMMA parameter used in lifting procedure
    @return None: The entire operation is conducted in place.
    @see: DWT97Rows, StandardProcedure, ISaveData
*/
__global__ void InverseDwt97(float *data, int width, int height,float coeff1,float
    coeff2){
    int col_idx = blockIdx.x * blockDim.x + threadIdx.x;
    int row_idx = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = row_idx * width + col_idx;


    __syncthreads();
    if(col_idx < width && row_idx < height){

        //Undo Update
        if (col_idx % 2 == 0 && col_idx < (width - 1) && col_idx > 0) {
            data[idx] += (-coeff2 * (data[(idx-1)] + data[(idx+1)]));
        }

        if (col_idx %2 == 0 && col_idx == 0) {
            data[idx] += (-coeff2 *(data[(idx+1)]));
        }

        __threadfence();

        //Undo predict
        if (col_idx % 2 == 1 && col_idx < (width - 2)) {
            data[idx] += (-coeff1 * (data[(idx-1)] + data[(idx + 1)]));
        }

        if (col_idx %2 == 1 && col_idx == (width-1)) {
            data[idx] += (-coeff1 * (data[idx-1]));
        }

        __threadfence();}}

/*
    This function has the purpose to invert the normalization step

    In constrast with the forward Normalization step, which is performed as the last step
         of direct procedure,
    the INormalization function is performed as first step to achive the inversion
        through a reverse ordering of the
```

```
    operations. This is achived by dividing back the even and odd elements by the K and
        IK normalization coefficients.

    @param data: Vectorized version of the original image expressed in float elements
        vector.
    @param width: Width of the original bidimensional matrix.
    @param height: Height of the original bidimensional matrix.
    @return None: The entire operation is conducted in place.
    @see Inversedwt97, StandardProcedure, Normalization
*/
__global__ void INormalization(float *data,int width,int height){
    int col_idx = blockIdx.x * blockDim.x + threadIdx.x;
    int row_idx = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = row_idx * width + col_idx;
    __syncthreads();
    if(col_idx < width && row_idx < height){

        if (col_idx % 2 == 0) {
            data[idx] /= K;
        }

        if (col_idx % 2 == 1) {
            data[idx] /= IK;
        }
    }}


/*
    Saving elaborated data in a quadrant style result.

    This step is not directly related with the lifting approch but is fundamental to
        ensure a meaningful rappresentation
    of the results produced. After the elaboration of alongside rows/cols though this
        procedure we are able to split
    even and odd elements in two different regions of the output and after repeating the
        process on the other axis to
    obtain a four quarters decomposition of the image.

    To observe the decomposition produces is possible to run the executable and
        visualizing it into ./Results folder.

    @param data: Vectorized version of the original image expressed in float elements
        vector.
    @param temp: Used to store the result produced by the splitting, need to be copied
        back to the original data vector.
    @param width: Width of the original bidimensional matrix.
    @param height: Height of the original bidimensional matrix.
    @return The entire operation is conducted using values of data and storing them into
        temp vector.
    @see ISaveData,DWT97Rows,StandardProcedure
*/
__global__ void SaveData(float *data,float*temp,int width,int height){
    int col_idx = blockIdx.x * blockDim.x + threadIdx.x;
    int row_idx = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = row_idx * width + col_idx;
    if(col_idx<width && row_idx < height){
        if(col_idx% 2 == 0){
            temp[((row_idx*width) + (col_idx/2))] = data[idx];
        }else{
            temp[(((width)/2) + ((row_idx*width) + (col_idx/2)))] = data[idx];
        }}}


/*
    Recompacting the decomposition produced by SaveData into the final image

    Since the elaboration done by the forward pass and the decomposition of SaveData
        produced a quadrants image
```

```
    is now necessary to perform the inverse procedure. This function performs the first
        step of the inverse routine
    and the effect that it produces is the actual compacting of odd and even elements
        into adjacent locations. By
    repeating this step BEFORE each InverseDWT97 function call and INormalization we
        ensure a correct working of the
    algorithm.

    @param data: Vectorized version of the original image expressed in float elements
        vector.
    @param temp: Used to store the result produced by the compacting step, need to be
        copied back to the original data vector.
    @param width: Width of the original bidimensional matrix.
    @param height: Height of the original bidimensional matrix.
    @return The entire operation is conducted using values of data and storing them into
        temp vector.
    @see SaveData, InverseDWT97, StandardProcedure
*/
__global__ void ISaveData(float* data, float* temp, int width, int height) {
    int col_idx = blockIdx.x * blockDim.x + threadIdx.x;
    int row_idx = blockIdx.y * blockDim.y + threadIdx.y;

    if (col_idx < width / 2 && row_idx < height) {
        int data_idx = col_idx + row_idx * (width / 2);

        temp[data_idx * 2] = data[col_idx + row_idx * width];
        temp[data_idx * 2 + 1] = data[col_idx + row_idx * width + width / 2];
        }}

__global__ void transposeMatrix(float *input, float *output, int rows, int cols) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    //printf("Transpose colled\n");

    // Check if within the valid range
    if (row < rows && col < cols) {
        int input_index = row * cols + col;
        int output_index = col * rows + row;
        output[output_index] = input[input_index];
    }}

/*
    Wrapper function to perform direct DWT97 and taking the times of the execution

    This function is in charge to instantiate the blocks of threads used for the
        computation of the dwt97 in an
    adaptive manner. By using the StandardProcedure function call first alongside rows
        (0,0) then the cols (1,0)
    through a transposition of the matrix we can perform the steps of predict1, update1,
        predict2, update2, normalization
    and take the time of their execution.

    @param data: Vectorized version of the original image expressed in float elements
        vector.
    @param width: Width of the original bidimensional matrix.
    @param height: Height of the original bidimensional matrix.
    @see: StandardProcedure

*/
void DWTForward97(float *h_image, int width, int height) {

    struct GeneralInfo INFO;

    dim3 nThreadPerBlock(BLOCK_SIZE, BLOCK_SIZE);

    dim3 nBlocks((width/nThreadPerBlock.x)+((width%nThreadPerBlock.x) == 0 ? 0:1),
                 (height/nThreadPerBlock.y)+((height%nThreadPerBlock.y) == 0 ? 0:1));
```

```
    cudaEvent_t start_direct, stop_direct;
    cudaEventCreate(&start_direct);
    cudaEventCreate(&stop_direct);

    INFO.nThreadPerBlock = nThreadPerBlock;
    INFO.nBlocks = nBlocks;
    INFO.width = width;
    INFO.height = height;
    size_t size = width*height*sizeof(float);

    float *d_image;

    cudaMalloc((void **)&d_image, size);

    cudaMemcpy(d_image, h_image, size, cudaMemcpyHostToDevice);

    cudaEventRecord(start_direct);
    StandardProcedure(d_image,&INFO,0,0);
    StandardProcedure(d_image,&INFO,1,0);
    cudaEventRecord(stop_direct);

    memset(h_image,0,size);
    cudaMemcpy(h_image, d_image, size, cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();
    cudaEventSynchronize(stop_direct);
    cudaDeviceSynchronize();


    float time_direct = 0;
    cudaEventElapsedTime(&time_direct,start_direct,stop_direct);

    printf("time for direct: %8.2f s\n",time_direct);

    cudaFree(d_image);
}

void DWTInverse97(float *h_image,int width,int height){
    struct GeneralInfo INFO;

    dim3 nThreadPerBlock(BLOCK_SIZE, BLOCK_SIZE);

    dim3 nBlocks((width/nThreadPerBlock.x)+((width%nThreadPerBlock.x) == 0 ? 0:1),
                 (height/nThreadPerBlock.y)+((height%nThreadPerBlock.y) == 0 ? 0:1));

    cudaEvent_t start_inverse,stop_inverse;
    cudaEventCreate(&start_inverse);
    cudaEventCreate(&stop_inverse);



    INFO.nThreadPerBlock = nThreadPerBlock;
    INFO.nBlocks = nBlocks;
    INFO.width = width;
    INFO.height = height;
    size_t size = width*height*sizeof(float);

    float *d_image;

    cudaMalloc((void **)&d_image, size);

    cudaMemcpy(d_image, h_image, size, cudaMemcpyHostToDevice);

    cudaEventRecord(start_inverse);
    StandardProcedure(d_image,&INFO,0,1);
    StandardProcedure(d_image,&INFO,1,1);
    cudaEventRecord(stop_inverse);

    memset(h_image,0,size);
```

```
    cudaMemcpy(h_image, d_image, size, cudaMemcpyDeviceToHost);
    cudaEventSynchronize(stop_inverse);

    float time_inverse = 0;
    cudaEventElapsedTime(&time_inverse,start_inverse,stop_inverse);

    printf("time elapsed for inverse: %8.2f s\n",time_inverse);

    cudaFree(d_image);
}

/*
    Parametrized routine to perform forward and inverse DWT97 transform on vectorized
        image.

    Since the steps conducted by the direct and inverse procedure of DWT97 share much
        similarities we implemented
    a common routine to perform them with different setting for each one of them.

    ___
    IMPORTANT: Since the image that we are working with is saved in vectorized form by
        rows to simplify the computation
    we opted for a trasposition of the matrix to perform the operation alongside the
        columns. The made choice is supported
    by an actual parallel implementation of the trasposition operation to avoid
        performance degradation at execution time.
    ____

    Cases of use:
        - Direct alongside rows: StandardProcedure(...,...,...,0,0)
        - Direct alongside cols: StandardProcedure (...,...,...,1,0) -> Transpose then
            dwt97Rows
        - Inverse alognside rows: StandardProcedure(...,...,...,0,1)
        - Inverse alongside cols: StandardProcedure(...,...,...,1,1) -> Transpose then
            Inversedwt97

    @param data: Vectorized version of the original image expressed in float elements
        vector.
    @param general_info: Struct made to include case-based variable such as width,height
        ,nThredPerBlockm,nBlocks.
    @param needsTranspose: Boolean variable to indicate if we need to traspose the
        matrix before performing the steps of lifting
                - 0: no transpose
                - 1: transpose
    @param inverse: Boolean variable used to switch between direct and inverse procedure
        .
                - 0: direct procedure
                - 1: Inverse procedure
    @return The entire operation is conducted using values of data and storing them into
         temp vector.
    @see DWTInverse97,DWTForward97

*/
void StandardProcedure(float *data,struct GeneralInfo *general_info, bool needsTranspose
    ,bool inverse){
    dim3 nBlocks = general_info->nBlocks;
    dim3 nThreadPerBlock = general_info->nThreadPerBlock;
    size_t size = general_info->width*general_info->height*sizeof(float);

    float * temp;
    cudaMalloc((void **) &temp,size);
    cudaMemcpy(temp,data,size,cudaMemcpyDeviceToDevice);

    if(needsTranspose){
        transposeMatrix<<<nBlocks,nThreadPerBlock>>>(data,temp,general_info->width,
            general_info->height);
        cudaDeviceSynchronize();
        cudaMemcpy(data,temp,size,cudaMemcpyDeviceToDevice);
```

```
        cudaMemset(temp,0,size);
    }
    if(inverse){
        ISaveData<<<nBlocks,nThreadPerBlock>>>(data,temp,general_info->width,
            general_info->height);
        cudaDeviceSynchronize();
        cudaMemset(data,0,size);
        cudaMemcpy(data,temp,size,cudaMemcpyDeviceToDevice);
        cudaMemset(temp,0,size);

        INormalization<<<nBlocks,nThreadPerBlock>>>(data,general_info->width,
            general_info->height);
        cudaDeviceSynchronize();
        InverseDwt97<<<nBlocks,nThreadPerBlock>>>(data,general_info->width,general_info
            ->height,GAMMA,DELTA);
        cudaDeviceSynchronize();
        InverseDwt97<<<nBlocks,nThreadPerBlock>>>(data,general_info->width,general_info
            ->height,ALPHA,BETA);
        cudaDeviceSynchronize();
    }else{
        dwt97Rows<<<nBlocks,nThreadPerBlock>>>(data,general_info->width,general_info->
            height,ALPHA,BETA);
        cudaDeviceSynchronize();
        dwt97Rows<<<nBlocks,nThreadPerBlock>>>(data,general_info->width,general_info->
            height,GAMMA,DELTA);
        cudaDeviceSynchronize();
        Normalization<<<nBlocks,nThreadPerBlock>>>(data,general_info->width,general_info
            ->height);
        cudaDeviceSynchronize();

        SaveData<<<nBlocks,nThreadPerBlock>>>(data,temp,general_info->width,general_info
            ->height);
        cudaDeviceSynchronize();
        cudaMemcpy(data,temp,size,cudaMemcpyDeviceToDevice);
        cudaMemset(temp,0,size);
    }
    if(needsTranspose){
        transposeMatrix<<<nBlocks,nThreadPerBlock>>>(data,temp,general_info->width,
            general_info->height);
        cudaDeviceSynchronize();
        cudaMemcpy(data,temp,size,cudaMemcpyDeviceToDevice);
        cudaMemset(temp,0,size);
    }

    cudaFree(temp);
}
```

Main for GPU algorithm – main.cu

```
#include <stdio.h>
#include "../Shared/utils.c"
#include "../Shared/compression_utils.c"
#include "DWT_gpu.cu"

int main(int argn,const char *argv[]) {

    if (argn < 2 || argn > 4) {
        printf("Usage: %s <file_path> <NeedsCompression> [<QuantizationStep> <
            ThresholdValue>]\n", argv[0]);
        return 1;
    }

    // The filepath is stored in argv[1]
    const char* filename = argv[1];
    // The flag is stored in argv[2]
    int flag = atoi(argv[2]);

    if (flag != 0 && flag != 1) {
        printf("Invalid flag. Flag must be either 0 or 1.\n");
        return 1;
    }

    if (flag == 1 && argn != 4) {
        printf("Flag is 1, but expected additional arguments are missing.\n");
        return 1;
    }

    // If flag is 1, we expect two additional arguments
    if (flag == 1) {
        // Process additional arguments
        int QuantizationStep = atoi(argv[3]);
        int Threshold = atoi(argv[4]);
    }

    int width, height;

    // Load grayscale image
    unsigned char* image = loadImage(filename, &width, &height);

    if (!image) {
        return 1; // Error loading image
    }

    // Convert the image to float for the transform
    float *floatImage = (float *)malloc(width * height * sizeof(float));
    for (int i = 0; i < width * height; i++) {
        floatImage[i] = (float)image[i];
    }

    size_t original_size_bytes = get_array_size_bytes(floatImage, width*height);

    // Perform the forward CDF 9/7 wavelet transform for a specified number of levels
    int levels=3;

    DWTForward97(floatImage,width,height);

    quantize_image(floatImage,width,height,10.0);
    threshold_image(floatImage,width,height,10.0);
    count_non_zero_elements(floatImage,width,height);
    CompressedSegment* compressed_data;
    int compressed_size;
    compress_image_rle(floatImage,width,height,&compressed_data,&compressed_size);

    printf("Original Size:%zu\n",sizeof(float)*width*height);
    printf("Compressed size:%zu\n",sizeof(compressed_data)*compressed_size);
    printf("Memory saved:%zu\n",(sizeof(float)*width*height) - (sizeof(compressed_data)*
```

```
        compressed_size));

    DWTInverse97(floatImage, width, height);

    free(floatImage);
    free(image);

    return 0;
}
```