


DWT transform

Wavelet transform using CUDA programming

Martina Roscica & Stefano Verrilli

University Parthenope

 github.com/StefanoVerrilli/DWT_CUDA

Wavelet transform

The Wavelet Transform stands out as a widely favored method for analyzing non-stationary signals and proves especially valuable for conducting multi-resolution signal analysis.

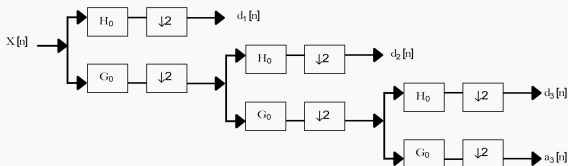
An essential feature of the wavelet transform is its ability to conduct an analysis that can be finely adjusted locally in both the time and frequency domains.

The wavelet transform can be customized by selecting from various families of wavelets. The simplest option is the Haar wavelet, while the most widely used is the Daubechies wavelet.

The Mallat algorithm

The traditional Discrete Wavelet Transform (DWT), conducted in the past year using the **Mallat algorithm**, involves breaking down signals using Finite Impulse Response (FIR) filters and then subsampling the outcomes in a sequential manner.

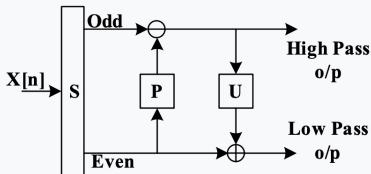
However, from a computational perspective, Mallat algorithm faces a significant drawback as it demands intensive processing, which conflicts with the requirement for rapid and real-time calculations.



Lifting Schema

In 1996, Sweldens defined a new approach to wavelet transform called **Lifting Scheme**. Lifting scheme is based on three steps:

- **Split:** Divide the input into odd and even elements.
- **Predict:** Approximate the odd samples by applying an interpolation function to the even one and add them together, this produces as output the high pass values.
- **Update:** Smooth the odd data and add them to the even samples to produce the low pass filtered values.



Daubechies Wavelet Transform

Selecting an appropriate wavelet family is a crucial task in Wavelet Transform applications. We will focus on the Daubechies Wavelet family, as it is widely used, with both the Daubechies 9/7 and 5/3 wavelets both employed in JPEG 2000.

Moreover, wavelet filters possess intriguing orthogonal properties and can be utilized for both lossy and lossless compression algorithms.

Sequential Implementation

Daubechies 9/7 details

The Daubechies 9/7 for 2D input is implemented by considering the lifting scheme version of the algorithm. This approach consider a four-step operations and a normalization step.

We will consider a sequence X and apply those operations independently on rows and cols:

$$\text{Predict 1 : } I_{2i+1} = X_{j,2i+1} + \alpha(x_{j,2i} + X_{j,2i+2}) \quad (1)$$

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

$$\text{Predict 2 : } P_{j-1,i} = I_{2i-1} + \gamma(I_{2i-2} + I_{2i}) \quad (3)$$

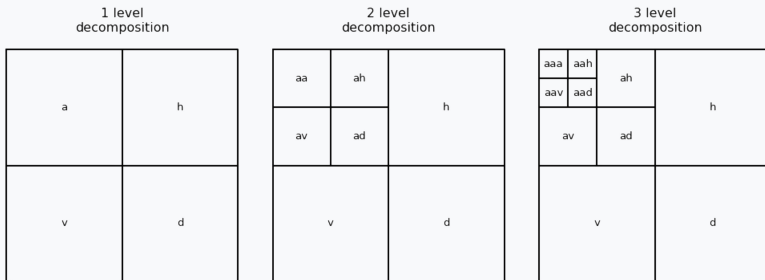
$$\text{Update 2 : } U_{j-1,i} = I_{2i} + \delta(P_{j-1,i} + P_{j-1,i+1}) \quad (4)$$

$$\text{Normalization 1 : } H_{j-1,i} = K * (P_{j-1,i}) \quad (5)$$

$$\text{Normalization 2 : } G_{j-1,i} = (1/K) * (U_{j-1,i}) \quad (6)$$

DWT results

The output is produced by applying the previously described lifting schema separately on rows and then on columns. This procedure is iterated by the levels required to modulate the details extracted.



Lifting steps applied

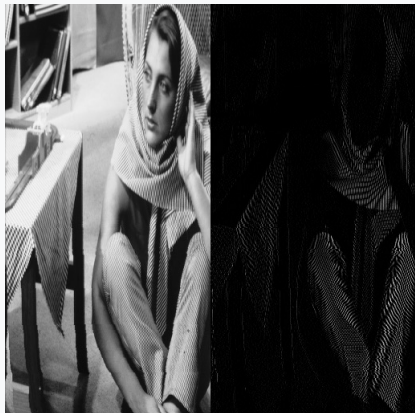


Figure: Rows/Cols application



Figure: Rows application

Implementation details

The implementation under consideration utilizes the C programming language for its computational efficiency and to facilitate potential GPU optimization.

Additionally, we integrated the **SDL2** external library, crucial for loading and saving post-processed images.

Finally, for the compression steps performed prior to the inverse reconstruction, we employed a variety of standard techniques that were implemented from scratch such as:

- Quantization
- Thresholding
- Run length encoding

Compression

To understand how this algorithm is involved in compression tasks, let's consider the previous image.

The following table summarizes results obtained after compression:

Parameters ¹	Original size ²	Compressed size	Saved space
20; 10	1048576	716296	332280
40; 20	1048576	445656	602920
60; 40	1048576	324496	724080

¹Represent Quantization Step and Threshold value

²Values are expressed in bytes.

Compressed images

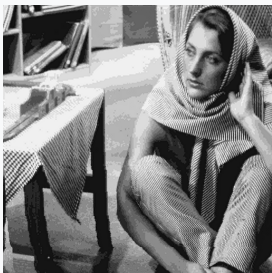


Figure:
reconstructed with
 $T=20$, $Q=10$.



Figure:
reconstructed with
 $T=40$, $Q=20$.



Figure:
reconstructed with
 $T=60$, $Q=30$.

Parallel Implementation

Why Lifting schema?

In addition to its computational benefits, one of the reasons why we have chosen to implement the wavelet transform using the lifting technique is its compatibility with GPU parallelization.

In fact, respect to Mallat approach based on convolution the lifting schema involves **fewer data movement** minimizing the memory access.

Moreover the steps of prediction and update can be computed in **saparate manner on different subsets of data** and permit the **fusion of multiple steps** in a single kernel function.

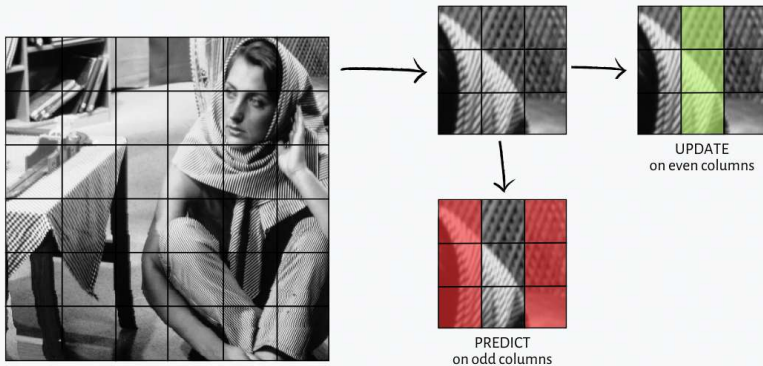
Parallelization utilized

To implement the operation of lifting using CUDA parallel computation we utilized a block-based technique to segment the image.

In particular, considering a section of the image of size $N \times N$, we **instantiated a number of threads equal to the number of pixel in a block.**

Through this particular instantiation of the kernel we made each thread responsible for updating a single element in each row/column.

Image segmentation and Lifting scheme



Synchronization details

To prevent misinterpretation regarding the predict and update steps of the lifting approach, specific synchronization mechanisms were thoughtfully implemented.

Specifically, the **__threadfence()** function was employed to ensure accurate substitution of values in the original image during both prediction and updating phases, as well as to guarantee proper data reading between these steps.

Furthermore, it was essential to incorporate thread synchronization using **cudaDeviceSynchronize()** after each subroutine of lifting following the execution of each kernel.

Code Implementation

```
// 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]));}
// Update Step
__threadfence();
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)]));}

dwt97Rows<<<nBlocks,nThreadPerBlock>>>(data,width,height,ALPHA,BETA);
cudaDeviceSynchronize();
dwt97Rows<<<nBlocks,nThreadPerBlock>>>(data,width,height,GAMMA,DELTA);
cudaDeviceSynchronize();
Normalization<<<nBlocks,nThreadPerBlock>>>(data,width,height);
cudaDeviceSynchronize();
```

Sequential and Parallel comparison

Advantages and Disadvantages

Methods	Pro	Cons
Sequential	<ul style="list-style-type: none">● Easily debuggable● Optimal (small sizes)	<ul style="list-style-type: none">● Slower computation● Unscalable● Missed multitasking
Parallel	<ul style="list-style-type: none">● Fast computation● Easily scalable● Lifting powered● Cuda based	<ul style="list-style-type: none">● Hardware intensive● Synchronization dependent● Nonparallelizable RLE

Table: Pro and cons of both approaches.

Execution time

To demonstrate the time benefits achieved by processing the algorithm in parallel versus sequentially, the execution times are:

Task	Sequential alg. ³	Parallel alg.
Direct DWT	30.0	0.00035
Inverse DWT	40.0	0.00046
Total	70.0	0.00081

³values are expressed in milliseconds.

Thank you for the attention!



https://github.com/StefanoVerrilli/DWT_CUDA