Accelerated Discrete Wavelet Transforms

E4750_2019Fall_ADWT_report

Kaylo Littlejohn kl3092, Desmond Yao jy2951
Columbia University Department of Electrical Engineering

Abstract

This work presents an improved version of the 2D discrete wavelet transform on GPUs using separable and nonseparable convolution based algorithm, aims to improve our solution by using shared memory, and is important given the wide spread of discrete wavelet transforms and the need for more efficient parallel solutions. The main challenge was developing an optimized separable or nonseparable algorithm and we overcame this by using a tiled schema. We accomplished our goal and found our tiled implementation performed up to two orders of magnitude better than our serial implementation and perform image compression as proof of concept.

1. Overview

1.1 Problem in a Nutshell

The discrete wavelet transform (DWT) is a signal processing technique used for extracting information and decomposing a signal into its low pass and high pass frequency components. 2D DWTs are most suitable for digital image processing applications such as image decompression or removing white Gaussian noise from an image. In image processing, DWTs decompose information into its low pass and high pass components, but since the DWT is a 1D transform being applied on both the horizontal and vertical directions of an image, this results in a two dimensional output (approximation, horizontal detail, vertical detail, and diagonal detail of the image).

To apply the DWT across both the horizontal and vertical directions of an image, we can either perform the DWT across the horizontal direction in the first GPU kernel and then across the vertical direction in a second GPU kernel or we can compute the result of the entire transform across the horizontal and vertical directions together. These two methods are known as separable DWTs and nonseparable DWTs respectively. The Cohen Daubechies Feauveau (CDF) wavelets are some of the most commonly used DWTs for image processing [12]. CDF wavelets come in the form of reversible or nonreversible wavelets, with reversible wavelets using only rational numbers and the non-reversible wavelets using irrational numbers, which results in rounding errors and hence is irreversible[8]. The later of the two CDF wavelets is known as CDF 9/7. Note that this does not mean that the DWT cannot be inverted for CDF 9/7 wavelets, but rather perfect reconstruction of the original image is not possible.

In this paper, we create implementations of separable and nonseparable 2D DWT algorithms for the CDF 9/7 wavelet using a PyCUDA framework. We optimize our separable implementation by loading our input arrays into shared memory and performing 2D DWTs using a tiled schema. We tested our code by performing the 2D DWT on a series of 750 images of varying height and width and explored the effects of varying the block dimensions of the input kernels when testing our scripts on random array.

1.2 Prior Work

A number of projects have done work involving 2D DWTs on GPUs. Tenllado et al. has compared the performance of two popular schemes known as the Filter Bank Scheme (FBS) and Lifting Scheme (LS) [5]. Our project implements the FBS, which has been shown by Tenllado et al. to outperform LS in current-generation GPUs [5], although Galiano et al. have written an in depth comparison paper between other and found conflicting results for the superior algorithm [7]. The FBS involves a direct convolution of the high pass and low pass filters of a 2D DWT, which are described in mathematical terms later in accordance to multiresolution signal decomposition textbook by Mallat, S [9] and is discussed in detail in section 2.1.

Kucis et al. used the CDF 9/7 wavelet in their 2D DWTs using the separable method in an OpenCL parallel programming framework [6] and Blazewicz et al. performed similar 2D DWTs on JPEG images for lossy image compression [8]. Furthermore, PyWavelets is an open source serial implementation for 2D DWTs created by Gregory et al. that we have found useful for producing serial results to compare our outputs with [4]. Lastly, Rout S. has provided a textbook on wavelets for image compression that we have found useful for determining differences between traditional orthogonal wavelets and biorthogonal wavelets such as the CDF 9/7 [3]. The same can be said of Gomes et al. in their biorthogonal wavelets textbook [2].

Despite the substantial amount of work being done on wavelet analysis, we did not come across any source code utilizing shared memory on GPUs.

2. Description

We provide a PyCUDA framework for implementing the separable and nonseparable 2D DWT transforms. We then extended our separable implementation to optimize execution time via shared memory. We first enumerate the objectives and technical challenges associated with generating CDF 9/7 wavelets, creating our filters, differentiating the requirements for the separable from nonseparable algorithms, and optimizing our naive separable implementation for shared memory. Next, we give a detailed description of our implementations and software design.

2.1. Objectives and Technical Challenges

Firstly, we must correctly generate our filters for the CDF 9/7 wavelet such that a convolution between our filters and the input arrays result in a 2D DWT. This is challenging because although the separable 2D DWT can rely directly on previously defined coefficients to generate 1D low pass and high pass filters, the nonseparable 2D DWT requires use to construct a series of four 2D filters based on arithmetic operations performed on the 1D CDF 9/7 coefficients [5].

Secondly, we must design and perform the horizontal and vertical convolutions in separate kernels for the separable 2D DWT implementations. This can be difficult because 2D DWTs produce output dimensions that are approximately half the size of the dimensions of the original input [10]. In nonseparable implementations, we still need to account for the difference in input to output size, but only a single kernel is required.

Thirdly, we need to expand our design to incorporate shared memory and experiment with different block sizes. This is challenging because the indexes of the output arrays vary dependent on mask length, which necessitates careful consideration for the correct output indices when using shared memory and furthermore, we must ensure our solutions work for variety of tile sizes.

Lastly, the 2D DWT only works for a single array at a time. However, since our test set is a series of 750 true color images, we need to take each dimension of color into consideration when producing our final approximations of the original images.

2.2. Problem Formulation and Design

The CDF 9/7 is part of a family of biorthogonal wavelets, which means that the associate wavelet transform is invertible but not necessarily orthogonal. Instead of using a single orthogonal basis, a pair of dual biorthogonal basis functions is employed [2]. In the context of multiresolution analysis, the analysis coefficients c and d of the projection of the wavelet, which is used to compute the forward portion of the transform, is given by the inner product of the representation f and basis $\phi_{j,k}$ and $\psi_{j,k}$, where j indicates the specific projection space of the wavelet and k is the corresponding coefficient of the signal [3][9].

$$c_k^j = <f, \phi_{j,k}>$$
$$d_k^j = <f, \psi_{j,k}>$$

The generated coefficients represent the low pass and the high pass filters of the DWT. Since f is known lossy representation function, we can compute the necessary coefficients, and hence filters, of our problem space (see appendix for full description) [3]. From here, we must take our low pass and high pass 1D filters and convert them into 2D filters for our approximation, horizontal detail, vertical detail, and diagonal details [5]. This is done by taking the outer product of all combinations of two series of coefficients, which are c_k^j and d_k^j in our case. The result is four square matrices representing all the filters needed for the nonseparable implementation. Note that the separable implementation can rely on c_k^j and d_k^j alone.

From here, we compute the 2D DWT firstly based on a separable schema. Since the DWT is inherently a one dimensional operation, it's helpful to first define the DWT mathematically as

$$y_{lowpass}[n] = \sum_{k=-\infty}^{\infty} x[k]c_{2n-k} = (x * c) \downarrow 2$$
$$y_{highpass}[n] = \sum_{k=-\infty}^{\infty} x[k]d_{2n-k} = (x * d) \downarrow 2$$

where $y_{lowpass}$ and $y_{highpass}$ are the resultant filters and x is the input array, which represents the red, green, or blue part of an image in our case. Notice how the convolution outputs are subsampled by 2, which is appropriate since during the compression of an image, or any other DWT application, half of the frequencies of the signal have been removed. Therefore, due to Nyquist's rule, half of the samples can be discarded [2][3][10].

In the 2D DWT separable schema, we perform the DWT first across the horizontal dimensions of image. This results in two subbands of horizontally subsampled partial low pass filtered and high pass filtered outputs. We then apply the DWT across the vertical dimensions of both subbands to produce the final output approximation, horizontal detail, vertical detail, and diagonal detail images. The block diagram below describes our filterbank method of computing the separable 2D DWT, where *H* and *G* represent *c* and *d*. Note that LL, HL, LH, and HH correspond to the approximation, horizontal, vertical, and diagonal output coefficients respectively (Figure 1) [3].

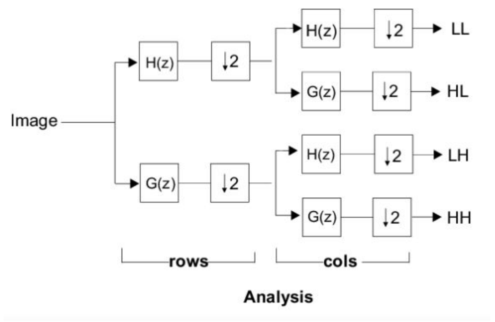


Figure 1 Separable 2D DWT Block Diagram:

Next, we implement a 2D DWT nonseparable schema. In this schema, we compute the convolution along the rows and columns of the images together using the 2D filters previously defined (Figure 2).

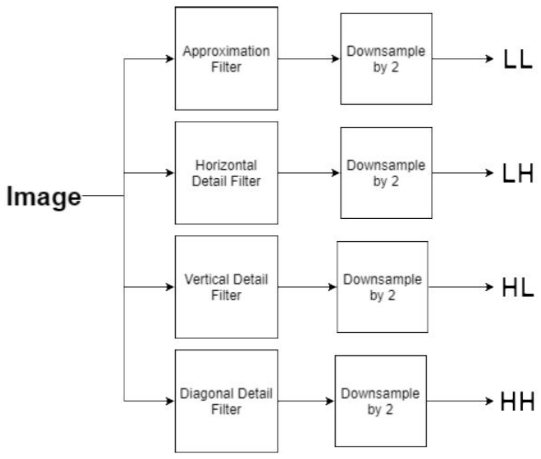


Figure 2: Nonseparable 2D DWT Block Diagram

Lastly, we optimized our separable schema, which was the faster algorithm of the two, by loading our input images and subbands into shared memory. In general terms, the separable schema is implemented as two separate kernels, with the first kernel receiving the input image and outputting the subbands, and the second kernel receiving the subbands and outputting the final compressed images. The nonseparable schema is implemented as a single kernel performing both dimensions of the convolution at once, however the input filters were an arithmetic combination of the original 1D CDF 9/7 coefficients as described. To produce a final output approximation image, we took the combination of the results of the 2D DWT applied along the red, green, and blue components of the input colored image. We also wrote a script which greyscales the input image and returns the greyscale approximation.

2.3 Software Design

Firstly, let's provide a high level overview of our 2D DWT implementation pipeline.

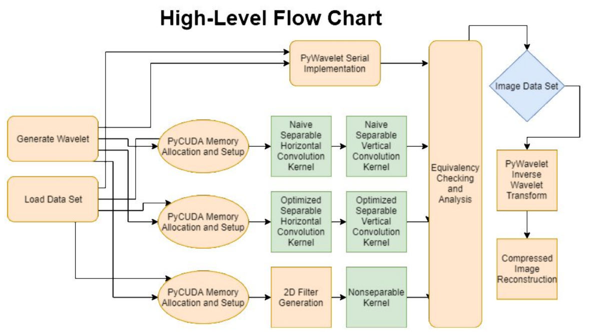
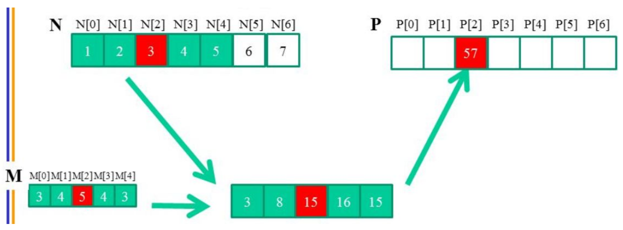
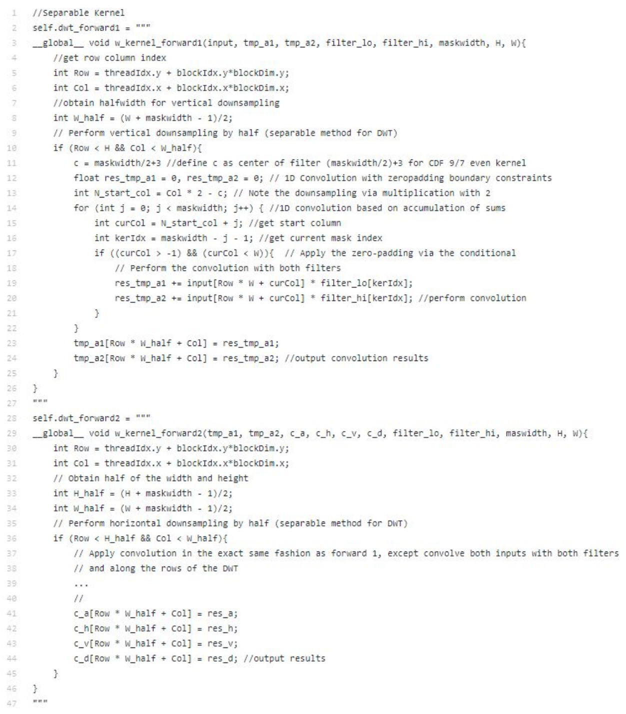

High-Level Flow Chart
Generate Wavelet
Load Data Set
PyWavelet Serial Implementation
PyCUDA Memory Allocation and Setup
Naive Separable Horizontal Convolution Kernel
Naive Separable Vertical Convolution Kernel
PyCUDA Memory Allocation and Setup
Optimized Separable Horizontal Convolution Kernel
Optimized Separable Vertical Convolution Kernel
PyCUDA Memory Allocation and Setup
2D Filter Generation
Nonseparable Kernel
Equivalency Checking and Analysis
Image Data Set
PyWavelet Inverse Wavelet Transform
Compressed Image Reconstruction

Figure 3 High Level Flow Chart of Implementation:

We can see from Figure 3 above that the input data is first generated either as a random signal or a series of 750 images. The random signals are useful because we can test a variety of block sizes for optimal performance without the need for running all 750 images multiple times on the GPU. We also generate our 1D convolution filters as an array of the CDF 9/7 wavelet coefficients (see Appendix). From here, we pass our input image into our serial, separable, nonseparable, and optimized kernels. Notice that the separable implementation has two kernels to complete the horizontal and vertical convolutions separately, however no memory is actually transferred to or from the GPU between kernel executions. Hence all operations for the separable implementation are still performed in parallel. After executing our kernels and serial implementation, all outputs are compared for equivalency and timing analysis is performed. Note that our serial implementation relies on the PyWavelets open source wavelet transform Python package [4]. The function is simple and takes in the wavelet and original signal and returns the approximation signal and its horizontal detail, vertical detail, and diagonal detail. We used PyWavelet's inverse 2D DWT function to reconstruct an example of compressed final image based on the corresponding approximation coefficients from any of the parallel or serial implementations and tested for equivalency across all output from all implementations [4].

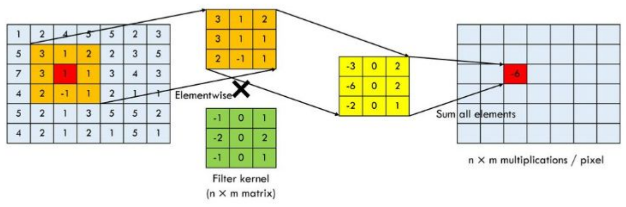

N
N[0] N[1] N[2] N[3] N[4] N[5] N[6]
1 2 3 4 5 6 7
P
P[0] P[1] P[2] P[3] P[4] P[5] P[6]
57
M
M[0]M[1]M[2]M[3]M[4]
3 4 5 4 3
3 8 15 16 15

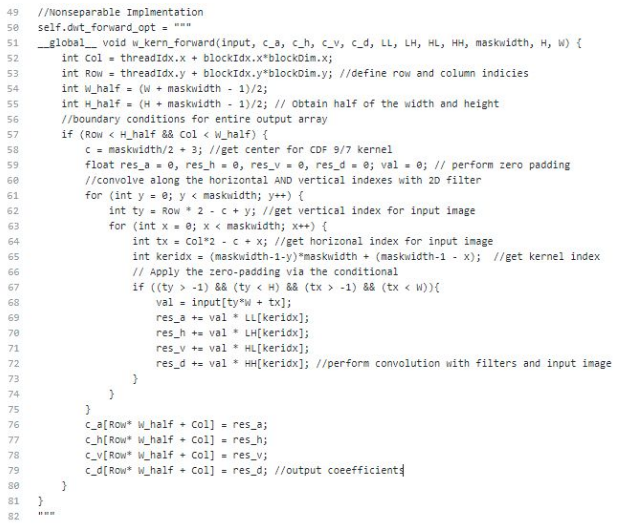


```
//Separable Kernel
self.dwt_forward1 = """
__global__ void w_kernel_forward1(input, tmp_a1, tmp_a2, filter_lo, filter_hi, maskwidth, H, W){
    //get row column index
    int Row = threadIdx.y + blockIdx.y*blockDim.y;
    int Col = threadIdx.x + blockIdx.x*blockDim.x;
    //obtain halfwidth for vertical downsampling
    int W_half = (W + maskwidth - 1)/2;
    // Perform vertical downsampling by half (separable method for DWT)
    if (Row < H && Col < W_half){
        c = maskwidth/2+3 //define c as center of filter (maskwidth/2)+3 for CDF 9/7 even kernel
        float res_tmp_a1 = 0, res_tmp_a2 = 0; // 1D Convolution with zeropadding boundary constraints
        int N_start_col = Col * 2 - c; // Note the downsampling via multiplication with 2
        for (int j = 0; j < maskwidth; j++) { //1D convolution based on accumulation of sums
            int curCol = N_start_col + j; //get start column
            int kerIdx = maskwidth - j - 1; //get current mask index
            if ((curCol > -1) && (curCol < W)){  // Apply the zero-padding via the conditional
                // Perform the convolution with both filters
                res_tmp_a1 += input[Row * W + curCol] * filter_lo[kerIdx];
                res_tmp_a2 += input[Row * W + curCol] * filter_hi[kerIdx]; //perform convolution
            }
        }
        tmp_a1[Row * W_half + Col] = res_tmp_a1;
        tmp_a2[Row * W_half + Col] = res_tmp_a2; //output convolution results
    }
}
"""
self.dwt_forward2 = """
__global__ void w_kernel_forward2(tmp_a1, tmp_a2, c_a, c_h, c_v, c_d, filter_lo, filter_hi, maswidth, H, W){
    int Row = threadIdx.y + blockIdx.y*blockDim.y;
    int Col = threadIdx.x + blockIdx.x*blockDim.x;
    // Obtain half of the width and height
    int H_half = (H + maskwidth - 1)/2;
    int W_half = (W + maskwidth - 1)/2;
    // Perform horizontal downsampling by half (separable method for DWT)
    if (Row < H_half && Col < W_half){
        // Apply convolution in the exact same fashion as forward 1, except convolve both inputs with both filters
        // and along the rows of the DWT
        ...
        //
        c_a[Row * W_half + Col] = res_a;
        c_h[Row * W_half + Col] = res_h;
        c_v[Row * W_half + Col] = res_v;
        c_d[Row * W_half + Col] = res_d; //output results
    }
}
"""
```

Figure 4: (top) 1D Convolution Diagram. (bottom) Separable 2D DWT Pseudocode.

The first kernel of the 2D DWT separable schema takes an input array, the input array dimensions, the lowpass CDF 9/7 analysis filter, and the highpass CDF 9/7 analysis filter. The output array is only half the size of the input array along the horizontal dimension, so we can reduce the grid size in the x dimension to the length of the horizontal portion of the input array. Then, we perform horizontal downsampling by initializing our convolution starting index as twice the traditional row index, hence removing half of the input samples [10][11]. We set a conditional such that the kernel only executes if our row is less than half the length of the input array, which prevents memory access errors since we need only half the threads along the x direction of the input array when working with horizontal downsampling. We then use a for loop to separately convolve the two filters with input array and generate our two subband outputs [10]. In Figure 4, we show an example of the 1D convolution along with some pseudocode for it's implementation in the context of a 2D DWT separable schema [13].



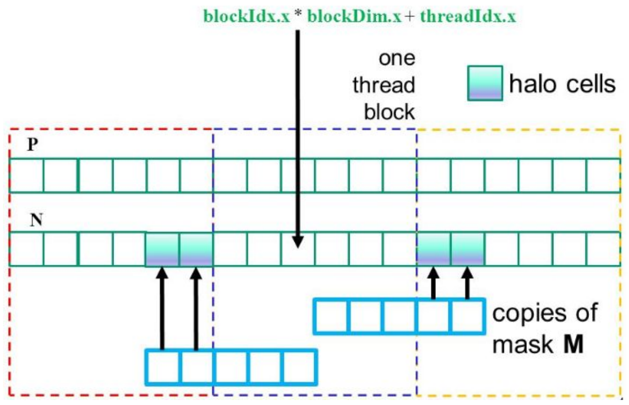


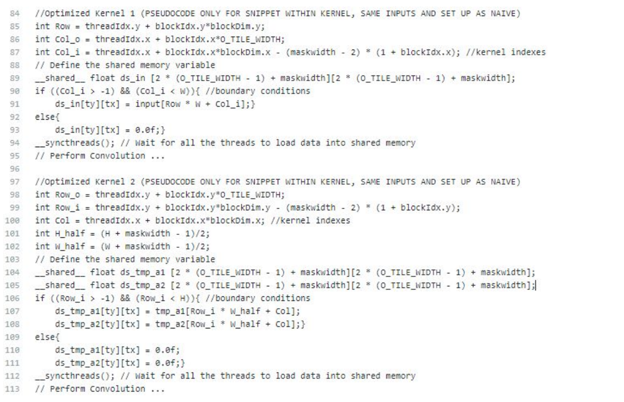
```
//Nonseparable Implmentation
self.dwt_forward_opt = """
__global__ void w_kern_forward(input, c_a, c_h, c_v, c_d, LL, LH, HL, HH, maskwidth, H, W) {
    int Col = threadIdx.x + blockIdx.x*blockDim.x;
    int Row = threadIdx.y + blockIdx.y*blockDim.y; //define row and column indicies
    int W_half = (W + maskwidth - 1)/2;
    int H_half = (H + maskwidth - 1)/2; // Obtain half of the width and height
    //boundary conditions for entire output array
    if (Row < H_half && Col < W_half) {
        c = maskwidth/2 + 3; //get center for CDF 9/7 kernel
        float res_a = 0, res_h = 0, res_v = 0, res_d = 0; val = 0; // perform zero padding
        //convolve along the horizontal AND vertical indexes with 2D filter
        for (int y = 0; y < maskwidth; y++) {
            int ty = Row * 2 - c + y; //get vertical index for input image
            for (int x = 0; x < maskwidth; x++) {
                int tx = Col*2 - c + x; //get horizonal index for input image
                int keridx = (maskwidth-1-y)*maskwidth + (maskwidth-1 - x);  //get kernel index
                // Apply the zero-padding via the conditional
                if ((ty > -1) && (ty < H) && (tx > -1) && (tx < W)){
                    val = input[ty*W + tx];
                    res_a += val * LL[keridx];
                    res_h += val * LH[keridx];
                    res_v += val * HL[keridx];
                    res_d += val * HH[keridx]; //perform convolution with filters and input image
                }
            }
        }
        c_a[Row* W_half + Col] = res_a;
        c_h[Row* W_half + Col] = res_h;
        c_v[Row* W_half + Col] = res_v;
        c_d[Row* W_half + Col] = res_d; //output coeefficients
    }
}
"""
```

Figure 5:(top) 2D Convolution Diagram. (bottom) Nonseparable 2D DWT Pseudocode.

The second kernel of the 2D DWT separable schema takes the two subbands from the first kernel, the input array dimensions, and the two CDF 9/7 analysis filters. Note that the the two subbands are not copied back onto the CPU, but rather remain on the GPU while transferring to the following kernel, thus no serial operations are involved. Then, in the same fashion, the two subbands are then convolved with the two CDF 9/7 filters, thus producing four output arrays: approximation coefficients, horizontal detail coefficients, vertical detail coefficients, and diagonal detail coefficients. In this case, both the x and y dimensions of the grid size can be reduced to the size of the output array, and the conditional to check for corresponding memory limits [10][11][13].

Note that in these cases, and the rest of our parallel implementations, a zero padding schema is involved. This means that at the boundary conditions, the convolution mask treats array elements at nonexistent indices as zeros. This is applied as a conditional statement to check whether or not the mask index is within the bounds of the indices of the input array. Furthermore, the 2D convolution requires that we flip our kernel before convolving, which we did outside of the kernel. We show pseudocode and a flow chart for this implementation in Figure 5 [13].





```
//Optimized Kernel 1 (PSEUDOCODE ONLY FOR SNIPPET WITHIN KERNEL, SAME INPUTS AND SET UP AS NAIVE)
int Row = threadIdx.y + blockIdx.y*blockDim.y;
int Col_o = threadIdx.x + blockIdx.x*O_TILE_WIDTH;
int Col_i = threadIdx.x + blockIdx.x*blockDim.x - (maskwidth - 2) * (1 + blockIdx.x); //kernel indexes
// Define the shared memory variable
__shared__ float ds_in [2 * (O_TILE_WIDTH - 1) + maskwidth][2 * (O_TILE_WIDTH - 1) + maskwidth];
if ((Col_i > -1) && (Col_i < W)){ //boundary conditions
    ds_in[ty][tx] = input[Row * W + Col_i];}
else{
    ds_in[ty][tx] = 0.0f;}
__syncthreads(); // Wait for all the threads to load data into shared memory
// Perform Convolution ...

//Optimized Kernel 2 (PSEUDOCODE ONLY FOR SNIPPET WITHIN KERNEL, SAME INPUTS AND SET UP AS NAIVE)
int Row_o = threadIdx.y + blockIdx.y*O_TILE_WIDTH;
int Row_i = threadIdx.y + blockIdx.y*blockDim.y - (maskwidth - 2) * (1 + blockIdx.y);
int Col = threadIdx.x + blockIdx.x*blockDim.x; //kernel indexes
int H_half = (H + maskwidth - 1)/2;
int W_half = (W + maskwidth - 1)/2;
// Define the shared memory variable
__shared__ float ds_tmp_a1 [2 * (O_TILE_WIDTH - 1) + maskwidth][2 * (O_TILE_WIDTH - 1) + maskwidth];
__shared__ float ds_tmp_a2 [2 * (O_TILE_WIDTH - 1) + maskwidth][2 * (O_TILE_WIDTH - 1) + maskwidth];
if ((Row_i > -1) && (Row_i < H)){ //boundary conditions
    ds_tmp_a1[ty][tx] = tmp_a1[Row_i * W_half + Col];
    ds_tmp_a2[ty][tx] = tmp_a2[Row_i * W_half + Col];}
else{
    ds_tmp_a1[ty][tx] = 0.0f;
    ds_tmp_a2[ty][tx] = 0.0f;}
__syncthreads(); // Wait for all the threads to load data into shared memory
// Perform Convolution ...
```

Figure 6:(top) 1D Tiled Convolution Diagram. (bottom) Optimized 2D DWT Pseudocode.

Our final kernel made use of the separable 2D implementation by loading all input data arrays in both kernels into shared memory. This way, the convolutions can be performed on a per-tile basis to determine the output approximation and detail coefficients. Here the second shared memory scheme from the lectures were applied where all the threads participate in loading data into shared memory but only some threads will compute the output. For the CDF9/7 wavelets, since the length of each filter is 10 samples, the input image is padded by 8 in all dimensions [13].

Then when performing the 1D convolution along the rows, first consider the first block and it's easy to see that each thread in the first block is tasked with loading in the pixel that's at the same row and is 8 pixels before the column of the thread. Because downsampling is also performed, the filter has a stride of 2 and thus the output tile has size $(block\_width, \frac{block\_width}{2} - 4)$. Then one way to assign the threads for writing the output pixels would be to let the first few consecutive threads that has column index less than the output tile dimension perform the computation, where each thread is writing the output corresponding to when the start of the flipped filter is placed at twice its thread column index.

Then for the i-th block, the j-th thread in each row of the second block should load in the pixel corresponding to the start of the flipped filter for the output it's trying to compute. Then the column index for the output pixel would be $(\frac{block\_width}{2} - 4) \times 2i + j$. Then since the array was padded by 8, the column index at the start of the filter would then be

$(\frac{block\_width}{2} - 4) \times 2i + j - 8 = i \times block\_width + j - 8(i+1)$

, which is the general formula for calculating the column index of the pixel each thread is supposed to load in.

When performing the 1D convolution along the columns, the same scheme can be used and the only necessary change would be to switch the rows and the columns. Our pseudocode for this optimized implementation is shown in Figure 6 alongside an illustration of the tiled 1D convolution [13].

Lastly, we tested our implementation on a variety of block and tile sizes so that we could determine the ideal kernel hyperparameters for faster execution times. For the full implementation and demo, visit our github link here: https://github.com/Kaylo1997/eecs4750project [1].

3. Results

In this section, we first briefly define the platform we used to implement our 2D DWTs, then include a detailed description of our results, and finally briefly describe how we verified our results.

3.1. Platform and Tool Description

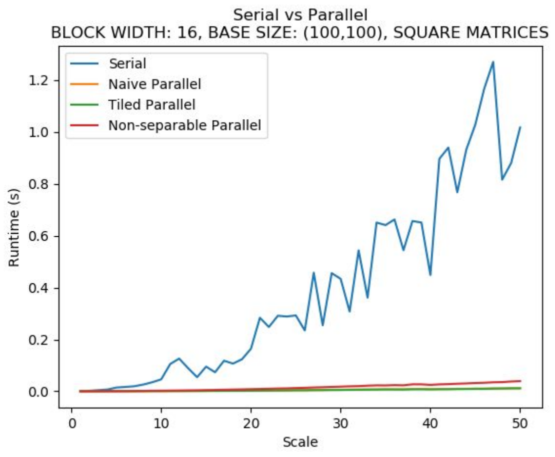
All of our results were tested on the Tesseract server equipped with an Intel Xeon E5-1620v CPU and an NVIDIA Tesla K40 GPU. The serial code was written using version 1.1.1 of the PyWavelets package [2] and the parallel kernels were written in CUDA and compiled using version 2019.1.2 of the PyCUDA package. Please see readme in GitHub link in section 2.3 for a detailed description of how to replicate our results.

3.2. Results Description

To compare the performance of the serial code with the various parallel implementations of DWT, we initially tested the runtime of all four programs on randomly generated Numpy arrays. In particular, the arrays started with either a base size of 100-by-100 to simulate square images, or a base size of 100-by-50 to simulate rectangular images and are scaled iteratively by a factor L that goes up to 50 so that the array size is either $100L \times 100L$ or $100L \times 50L$. In each iteration, the runtime for each individual program is recorded. In addition, we also experimented with the effect of the block dimension on runtime and used either block dimension of 16 or 32 for running the parallel kernels. It was found that regardless of the shape of the input and the block size, the parallel kernels all achieve up to two orders of magnitude faster runtime compared to the serial code. For the parallel kernels, it was found that both the naive separable and the tiled separable kernel achieved around 5-10 times faster runtime compared to the non-separable parallel kernel and the tiled separable kernel is the fastest of all parallel kernels. Additionally, it was also discovered that having a block size of 32-by-32 speed up the tiled separable kernel more compared to having a block size of 16-by-16.

We subsequently tested the performance of all programs using 750 images of various shapes (square, tall rectangular with ratio of 2-by-1, and wide rectangular with ratio of 1-by-2) and various sizes (size of the shortest side from 100 to 1000) and recorded the cumulative runtime for each program as the number of pixels processed increases. The reason we chose to experiment with block size with randomly generated numpy arrays was because we knew that varying block sizes for multiple iterations of 750 2D DWTs on images on the GPU would take up an enormous amount of time. This being said, we decided to test our implementation on random signals first to determine the ideal block size.

3.3. Figures



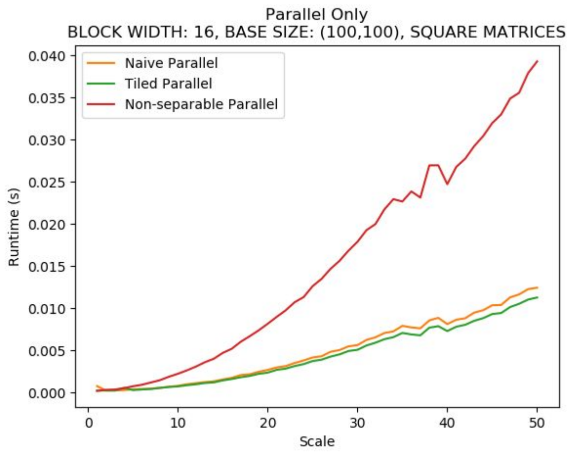


Figure 7: (top) Square Matrix Execution Time Comparison with Serial Implementation. (bottom) Square Matrix Execution Time Comparison without Serial Implementation. Block Width 16.

In Figure 7, the runtimes are plotted for all four programs and for the three parallel kernels when the input is a square array and the block width is 16. It can be seen that as the size of the input random array increases, the runtime of the serial code increases quadratically, while the runtimes of all parallel kernels increases almost linearly and are much smaller compared to the runtime of the serial code. When analyzing the performance of the parallel kernels individually, it can be seen that the non-separable kernel using 2D convolution is around 10 times slower compared to the two separable kernels and the tiled separable using shared memory is slightly faster compared to the naive separable kernel. In this case, the tiled separable kernel can be up to around 100 times faster than the serial kernel.

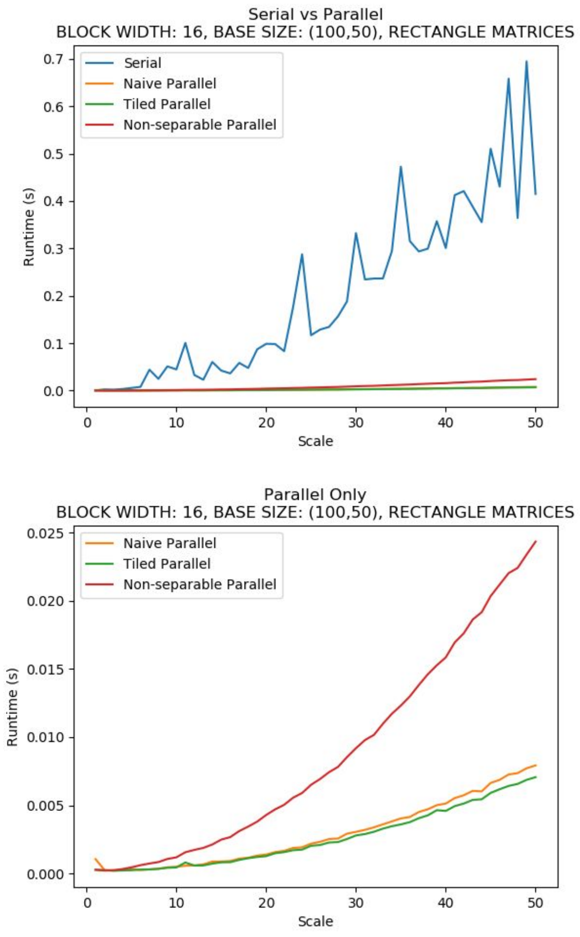


Figure 8: (top) Rectangle Matrix Execution Time Comparison with Serial Implementation. (bottom) Rectangle Matrix Execution Time Comparison without Serial Implementation. Block Width 16.

In Figure 8, runtimes are shown when the input shape is changed to be rectangular and the block width is still 16. Here the same pattern can be observed, where the serial code runtime increases quadratically and is much larger compared to the almost linearly-increasing parallel kernel runtime. For the parallel kernels, it can be seen that both the naive separable and the tiled separable kernel out performs the non-separable kernel by a factor of 5 and the tiled separable kernel is again slightly faster compared to the naive separable kernel.

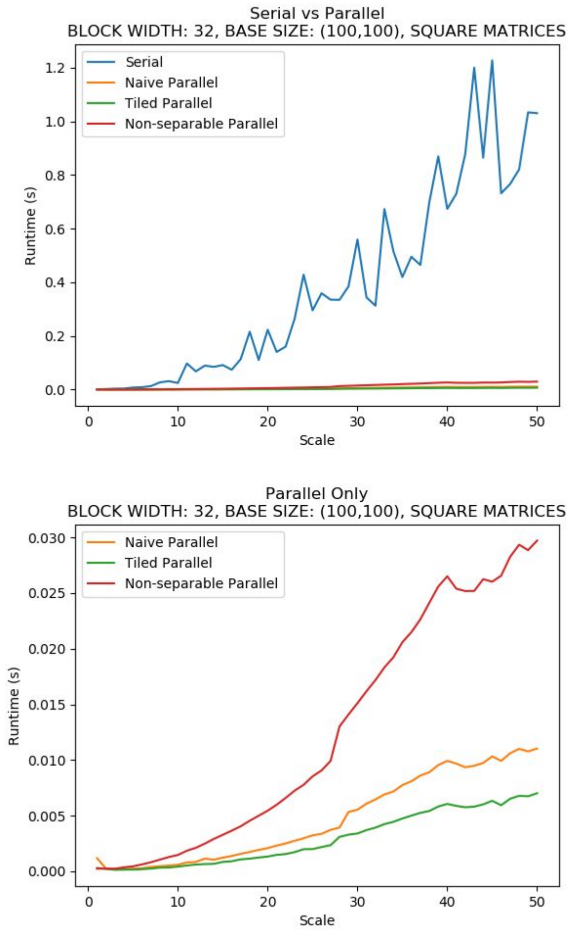
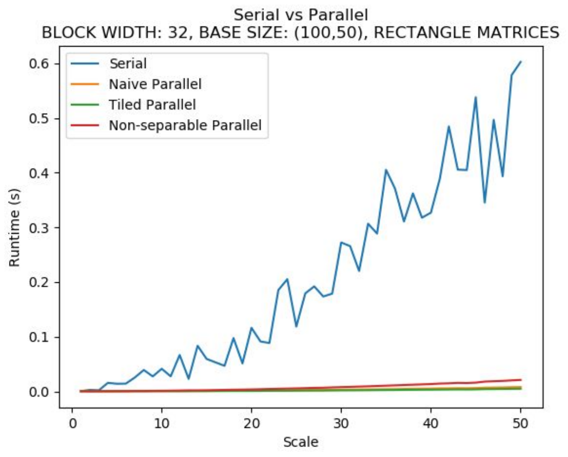


Figure 9: (top) Square Matrix Execution Time Comparison with Serial Implementation. (bottom) Square Matrix Execution Time Comparison without Serial Implementation. Block Width 32.

In Figure 9, runtimes are shown for square inputs when the block size is now changed to 32-by-32. Here the runtimes for all parallel kernels are still much faster compared to the serial code as expected. Furthermore, compared to Figure 7, the runtime of the tiled separable kernel is further reduced compared to when the block size was 16-by-16 by almost a factor of 2 while the runtime of the naive separable kernel remained largely the same. Additionally, the runtime of the non-separable kernel is also reduced slightly compared to when the block size was 16-by-16. Then here, using tiled separable kernel would ensure a runtime that’s around 200 times faster compared to the serial implementation of DWT.



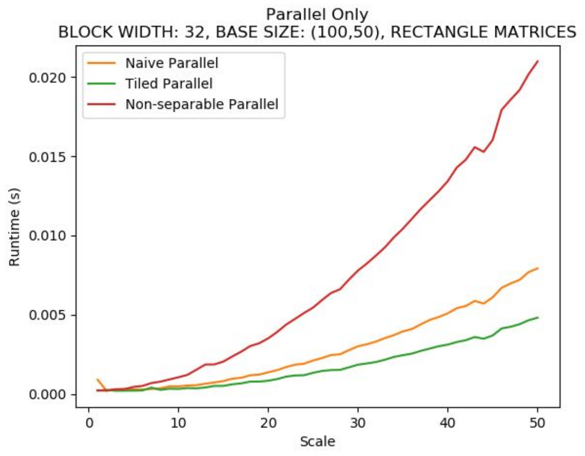
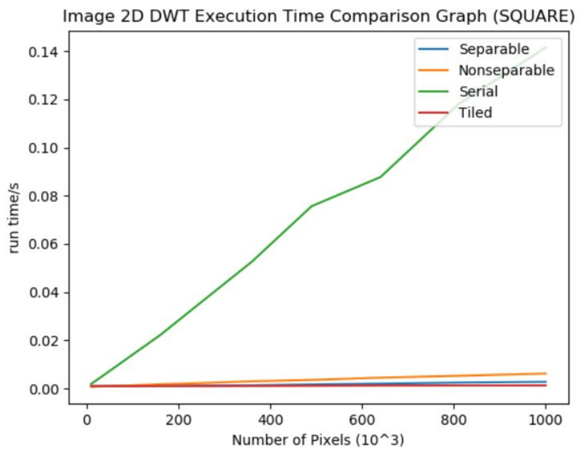
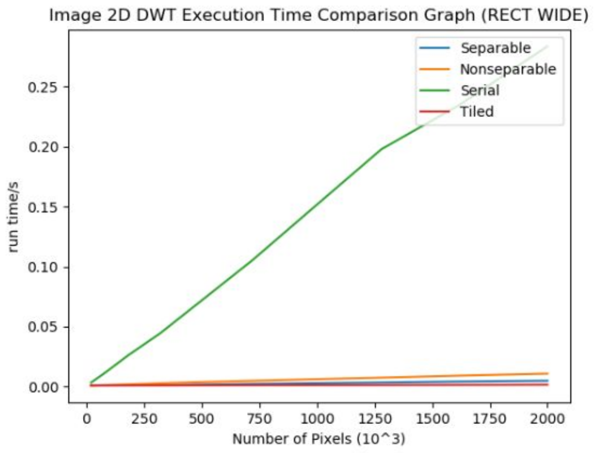


Figure 10: (top) Rectangle Matrix Execution Time Comparison with Serial Implementation. (bottom) Rectangle Matrix Execution Time Comparison without Serial Implementation. Block Width 32.

Lastly, in Figure 10, the execution times are given for rectangular inputs and block size of 32-by-32. Again, similar to the trend observed before for the square images, all parallel kernels are still much faster compared to the serial code. Additionally, increasing the block size decreased the runtime of the tiled separable kernel by almost a factor of 2 while the naive separable kernel is unaffected by the change. Furthermore, there is again a slight decrease for the runtime of the non-separable kernel even though it is still slower than both of the separable methods. In this case, using the tiled separable kernel would be around 200 times faster compared to the serial code.





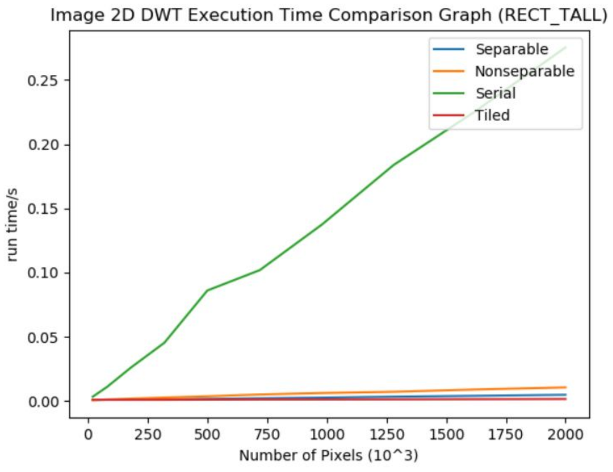


Figure 11: (top) Square Image 2D DWT Execution Time Comparison. (middle) Wide Rectangular Image 2D DWT Execution Time Comparison. (bottom) Tall Rectangular Image 2D DWT Execution Time Comparison.

When it comes to actual images, the same trend from before is still observed (Figure 11). Regardless of the shape of the input image, it can be seen that for the serial code, the time taken to compute the DWT for all the images increases as the size of all pixels processed increases, and the serial code runtime is again much higher compared to all the parallel kernels. For the parallel kernels, it can be seen that tiled separable kernel takes the least amount of time to process all the images and again achieves a two orders of magnitude speedup, while the non-separable kernel is again the slowest of all parallel kernels. This execution time plot yields the fastest times using a block size of 32, which we verified in Figures 7-10.

3.4. Test and Verification

To ensure the validity of our output, when using the random Numpy array to test the various different DWT implementations, during each iteration equality up to floating point error of the approximation and detail coefficients are tested between the serial code output and output from each parallel kernel separately and the program is designed to throw an error if inequality is detected during any iteration.

When generating the results using the actual images, we performed the same tests. However, this time we checked for equivalency for the coefficients for the red, green, and blue components of the image separately. Lastly, we demonstrate our previously verified correct outputs as described in the following section.

4. Demonstration

To demonstrate our 2D DWTs, we generated the approximation and detail coefficients using any one of our implementations. We then used PyWavelets inverse 2D DWT function to use our approximation coefficients to display a compressed approximation image (Figure 12) [4].

Figure 12a: Original Image

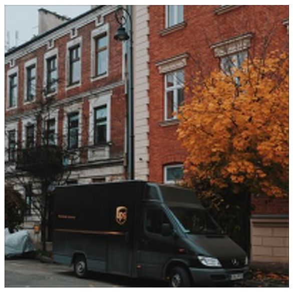


Figure 12b: Compressed Approximation Image

We also wrote a small test script which greyscales an image and then performs the 2D DWT using our serial implementation (Figure 13).

Figure 13a: Original Image

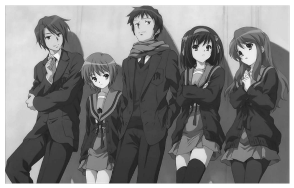


Figure 13b: Compressed Approximation Image

Notice how both images are the same size. This is expected since the inverse DWT restores the original dimensions of the image, except with half of the sample space removed and hence compressed.

5. Discussion and Further Work

From the above results, it follows our initial expectation that the various parallel implementations of DWT are all significantly faster compared to the serial implementation. Additionally, the fact that the tiled parallel kernel has the shortest runtime was also consistent with our expectations. Here it's interesting that changing the block size has a high impact on the tiled kernel. One potentially explanation is that when the block size is 16-by-16, the output tile size is 16-by-4 when doing the first forward pass and each block reads much more pixels than the amount it writes (16-to-4, ratio of 4:1). Whereas when the block size is 32-by-32, the output tile size is 32-by-12 and now the ratio between the read and the write is then (32-to-12, ratio of 2.67) and in general less amount of threads are wasted.

Previous literature reported in general that the nonseparable implementation is faster compared to the separable implementation, which is the opposite of what we observed [6][8]. From a detailed profile of our results using the NVIDIA Visual Profiler, we can see that our nonseparable implementation took up 67.5% of our kernel execution time, whereas the separable implementation took up only 30.25% of the execution time (Figure 14). One possible way we could improve our nonseparable implementation would be to incorporate a tiled schema into our 2D convolution algorithm, use constant memory for static variable with small memory requirements, and consolidate our code into the most efficient way possible (i.e. hardcoding a predetermined more ideal blocksize).

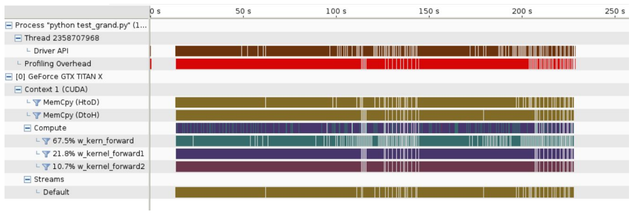


Figure 14: Profiler Results (see GitHub for full scale image)

Furthermore, the horizontal convolution kernel took approximately 21.8% of the total kernel execution time whereas the vertical convolution only took up 10.7% of the kernel execution time. This can be explained by the amount of calculation that's required as the input has not been downsampled yet for the horizontal convolution. But for the vertical convolution the input arrays have already been downsampled and thus the amount of calculation required is reduced in half.

We also observed that our program took a total of roughly 16 seconds to copy 40 GB of data onto the GPU but only 1.5 seconds to copy 11 GB back onto the CPU. It is expected that we will copy more memory to the GPU than from the GPU, yet perhaps in the future we could improve our algorithm by devising a schema which avoids copying many empty arrays onto the GPU.

Furthermore, our implementation shows inactivity on the GPUs streaming multiprocessors during the first 20 seconds of runtime, which is somewhat expected since we are loading in a data set of 750 images, yet because the images are less than 2.0 MB in total size, perhaps we could improve our code by loading in our data set using a single grand loop rather than a complicated piecewise loading schema.

We could also improve our implementation by finding ways to increase the amount of time performing computation compared to the amount of time required for copying memory, increasing utilization of host to device bandwidth during memory copying, or verifying that all of the multiprocessors of the GPU of choice are being utilized. We could also improve our model by experimenting with running memory copies and kernels in parallel, which is supported for NVIDIA GPU architectures [13]. Additionally, another interesting thing to experiment with would be nonseparable parallel kernel using shared memory and compare its performance with tiled separable parallel kernel.

However, despite areas for improvement, our optimized version of the separable algorithm performed far better than both the naive separable and nonseparable algorithms and is a new schema designed by ourselves with the aid of Kirke et al.'s slides [13]. Therefore, our final results are consistent with the previous literature and research being done regarding GPU performance optimization and 2D DWTs [5][6][7][8][13].

6. Conclusion

In conclusion, our project succeeded in developing an optimized version of a 2D DWT using shared memory. We also succeeded in creating necessary separable and nonseparable 2D DWT implementations. We saw that in comparison with serial methods of 2D DWTs, parallel 2D DWTs performed far better than CPU based implementations, with our optimized version performing up to two orders of magnitude better.

We learned that GPU programming can be a very effective way of speeding up computation for important parallelizable image processing applications, most notable image compression. We saw how any image can be compressed far faster using a GPU than a CPU, and that execution times improve quadratically as the original image becomes. In the future, we can explore improving our model by employing more constant memory wherever appropriate to avoid memory access delays due to using global memory.

7. Acknowledgements

We would like to acknowledge Pierre Paleo and the Julia Language Program for their help and guidance on how to compute 2D DWTs [10,11]. Furthermore, we would like to acknowledge David B. Kirke, Wen-Mei W. Hue, and NVIDIA for their images and help with understanding how to compute 1D and 2D convolutions [13]. Lastly, we would like to thank Gregory R. Lee et al. for their open sourced PyWavelets serial 2D DWT implementation [4].

8. References

[1] Accelerated Discrete Wavelet Transforms project GitHub: https://github.com/Kaylo1997/eecs4750project

[2] Gomes J., Velho L. (2015) Biorthogonal Wavelets. In: From Fourier Analysis to Wavelets. IMPA Monographs, vol 3. Springer, Cham

[3] Rout, S. (2003). Orthogonal vs. Biorthogonal Wavelets for Image Compression.

[4] Gregory R. Lee, Ralf Gommers, Filip Wasilewski, Kai Wohlfahrt, Aaron O'Leary (2019). PyWavelets: A Python package for wavelet analysis. Journal of Open Source Software, 4(36), 1237, https://doi.org/10.21105/joss.01237.

[5] Tenllado, C., Setoain, J., Prieto, M., Piñuel, L., & Tirado, F. (2008). Parallel implementation of the 2D discrete wavelet transform on graphics processing units: Filter bank versus lifting. IEEE Transactions on Parallel and Distributed Systems, 19(3), 299-310.

[6] Kucis, M., Barina, D., Kula, M., & Zemcik, P. (2014, October). 2-D discrete wavelet transform using GPU. In 2014 International Symposium on Computer Architecture

and High Performance Computing Workshop (pp. 1-6). IEEE.

[7] Galiano, V., López, O., Malumbres, M. P., & Migallón, H. (2013). Parallel strategies for 2D discrete wavelet transform in shared memory systems and GPUs. The Journal of Supercomputing, 64(1), 4-16.

[8] Błażewicz, M., Ciżnicki, M., Kopta, P., Kurowski, K., & Lichocki, P. (2011, August). Two-dimensional discrete wavelet transform on large images for hybrid computing architectures: GPU and CELL. In European Conference on Parallel Processing (pp. 481-490). Springer, Berlin, Heidelberg.

[9] Mallat, S. (1999). A wavelet tour of signal processing. Elsevier. Mallat, S. (1989). A theory for multiresolution signal decomposition: the wavelet representation. IEEE Transactions on Pattern Analysis & Machine Intelligence, (7), 674-693.

[10] Paleo, P. Parallel DWT source code. Github: https://github.com/pierrepaleo/PDWT

[11] Julia: A Fresh Approach to Numerical Computing. Jeff Bezanson, Alan Edelman, Stefan Karpinski and Viral B. Shah (2017) SIAM Review, 59: 65–98. doi: 10.1137/141000671. url: https://julialang.org/research/julia-fresh-approach-BEKS.pdf. JuliaParallel CUDA DWT source code. Github: https://github.com/JuliaParallel/rodinia

[12] Cohen, A., Daubechies, I., & Feauveau, J. C. (1992). Biorthogonal bases of compactly supported wavelets. Communications on pure and applied mathematics, 45(5), 485-560.

[13] Kirke, D., Hue, W. NVIDIA and University of Illinois Accelerated Computing Course.

9. Appendices

Data Sheet for the CDF 9/7 Wavelet Analysis Coefficients (zero padded for highpass filter) [10][12]:

k	Lowpass	Highpass
-4	0.02674875741	0
-3	−0.016864118443	0.09127176311
-2	−0.078223266529	−0.057543526229
-1	0.2668641184	−0.591271763114
0	0.6029490182	1.11508705
1	0.2668641184	−0.591271763114
2	−0.078223266529	−0.057543526229
3	−0.016864118443	0.09127176311
4	0.02674875741	0

Appendix I: CDF 9/7 Analysis Filter Coefficients

Individual Student Contributions (in %)

Task	**kl3092**	**jy2951**
Overall	**50**	**50**
Serial	**0**	**100**
Separable	**0**	**100**
Nonseparable	**95**	**5**
Shared Memory	**40**	**60**
Serial Code, Testing Scripts, and Analysis	**50**	**50**
Proposals and Background	**50**	**50**
Report	**55**	**45**