CCE3015 - Assignment 1 – Problem Research and Planning

Graham Pellegrini

November 8, 2024

1 Question 1

The Multi-level 3D Discrete Wavelet Transform (DWT) is an effective technique in signal processing, particularly useful for data compression and denoising applications. This transform decomposes a 3D dataset into sub-bands, capturing frequency components across multiple resolutions. Each decomposition level separates the data into approximation and detail coefficients, progressively breaking down the data into low and high frequency componenets.

1.1 Problem Overview

In this problem, we will be working with the CHOAS dataset [**?**], which contains large 3D medical images saved as slices in DICOM (.dcm) files. Given the dataset's size and format, these slices are first pre-processed in Python to convert them into a binary (.bin) file. The preprocessing pipeline involves reading DICOM files, transforming them into NumPy arrays, and saving the arrays as binary files. This binary format will allow for efficient loading and processing in the C++ implementation of the 3D DWT.

In the preprocessing step, to achieve a 3D volume from 2D slices. The slices are stacked along the depth axis to form a 3D volume, as shown in 1above. This 3D volume is saved as a numpy array in Python and then converted to a binary file for input into the C++ program. Like this the dicom slices have been converted into their respective 3D image as a volume.

1.2 Algorithm Overview

To implement the 3D DWT in C++, we will follow these key steps:

Input/Output Handling - Two utility header files will be included, one to load the binary data into a 3D volume array and another to save the processed data back into a binary file. These functions will facilitate data handling

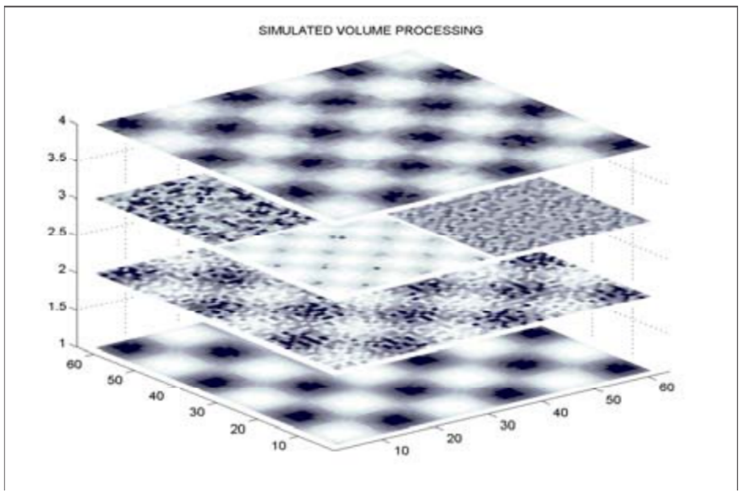

Figure 1: 3D Volume from 2D Slices [**?**]

between Python and C++.

Wavelet Coefficients - Daubechies wavelet filters will be used, taking from db1 till db4. These filter coefficients will directly be sampled from pywavelets, for comparison purposes.These coefficients will be hardcoded as vectors with seperation between low-pass and high-pass filters. The filters will be applied to the 3D data in the convolution step. Having the low filters and high filters seperated will allow for easy implementation of the convolution step. The low emphasise the approximation coefficients and the high emphasise the detail coefficients of the signal.

Discrete Wavelet Transform (DWT):

The pipeline shown in 2 shows how the 3D volume can be spilt into 8 subbands. However first a function to perform the 1D DWT on a signal will be defined. This function will take the signal and convolve it with the low-pass and high-pass filters. The two seperated convolved otuputs will then be downsampled by selecting every second value. The downsampling technique is there as a way to reduce the data size by half along each axis as well as a data redundancy reduction technique.

This 1D DWT function will then be called three times in the 3D DWT function, as it will be applied to each axis of the 3D volume. The 3D DWT function will take the 3D volume and apply the 1D DWT function to each axis. To keep convention the dwt is first applied to the column axis and the respective L and H subbands will be stored back to the original volume. Then the dwt is applied to the row axis and now we get the LL LH HL HH subbands. Finally the dwt is applied to the depth axis and we get the 8 subbands (LLL LLH LHL LHH

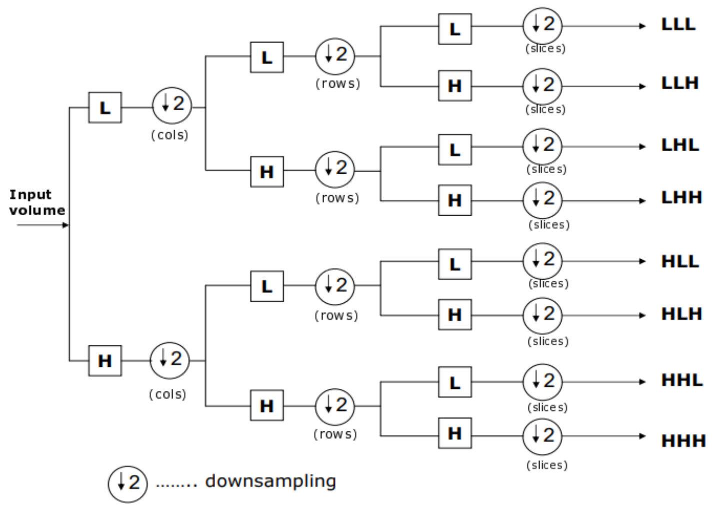

Figure 2: Discrete Wavelet Transform [?]

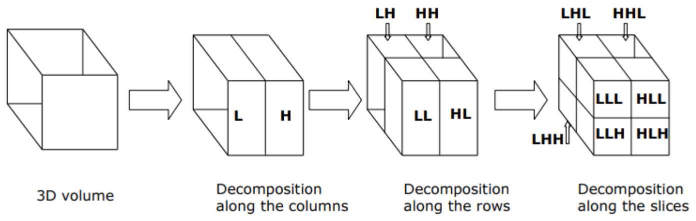
3D volume
L
H
Decomposition along the columns
LH
HH
LL
HL
Decomposition along the rows
LHL
HHL
LLL
HLL
LLH
HLH
LHH
Decomposition along the slices

Figure 3: 3D DWT Subbands [?]

HLL HLH HHL HHH). So the order 3D DWT fucntion will include 3 nested for loops to apply the respective column-wise, row-wise and depth-wise DWT.

The importance of always saving back the L and H subbands to the original volume is so we utalise memory efficiently, especially since we are working with 3d vectors of a large dataset.

Multi-level Transform - Our implementation aims at achieving Multi-level 3D DWT. This means we can take the approximation coefficients (LLL) from the first level and apply the 3D DWT again to get the approximation coefficients of the next level. This process can be repeated for as many levels as desired. However, it is important to note that while there is no theoretical limit to the number of levels, practical constraints are imposed by the image dimensions. In our case since the slices have dimensions of 78x256x256, a 4-level decomposition results in dimensions of 10x64x64, beyond which further decomposition would degrade image quality excessively.

A multi-level function will be defined that will call the 3D DWT function for the number of levels specified. If the level is not the final one then the volume dimensions will be halved along each axis, taking the approximation coefficients (LLL) and applying the 3D DWT function to them. In this process the higher level detail coefficients are discarded but they are not needed.

To be able to handle odd dimensions, either the dimention division must handle odd numbers or the volume must be padded with zeros to make the dimensions even.

1.3 Evaluation and Testing

To evaluate the implementation, we will have a post-processing step in Python. The saved output binary file will be loaded back into Python, and the 3D volume will be reconstructed from the subbands. This reconstructed coefficients will be compared to those achieved on the same volume with pywavelets. The comparison will be both visual and quantitative, using metrics like Mean Squared Error (MSE) and Eucaledian Distance. Although, we will be using the same coefficients as pywavelets, there might still be some differences in implementation. Such as the convolution, downsampling techniques and padding techniques. So the graphical comparison will be the main objective and then the statistical comparison will be a bonus to see how close the implementation is to the pywavelets implementation.

2 Question 2

To plan an efficient parallel implementation of the 3D Discrete Wavelet Transform (DWT) using CUDA, we analyze the operations that can be vectorized and optimized whilst handeling the issue of synchronization across the architecture.

2.1 Identifying Parallel Operations

In the DWT process, each dimension of the 3D volume undergoes convolution with wavelet filters. This process is currently implemented sequentially, as each dimension is processed one after the other. However, this can be parallelized by using thread blocks in CUDA to process each dimension concurrently. We note that in GPU programming, the number of threads per block and blocks in the grid is a crucial parameter that must be manually tested to find the optimal values for the particular GPU setup. At the same time, we must avoid overfitting to the GPU's capabilities as this may reduce performance from one system to another or even one dataset to another.

So the configured threading of the individual DWT in the 3D DWT function will be one step of the parallelization from which performance can benefit. Instead of processing individual elements of the 3D volume sequentially, we can use CUDA to launch kernels that handle the 1D DWT on different parts of the volume concurrently. This allows multiple elements of the volume to be accessed and processed in parallel, significantly improving performance.

Since we are parallelizing the data processing, we must revise the loops present to handle the parallel execution. By configuring the number of threads per block and blocks in the grid, we can ensure that the data is processed efficiently in parallel. This will reduce the overhead associated with sequential processing, thus enhancing efficiency.

The I/O operations, such as loading and saving to binary files, can also be parallelized. This involves reading the 3D volume's dimensions and allocating the required memory in the GPU's global memory. CUDA kernels can then be used to read and process the data from/to the binary file in parallel. After the kernel processes the data, it is copied back to the host memory, completing the loading or saving of the volume from/to the binary file.

2.2 Required Synchronization Points

In CUDA programming, synchronization points are necessary to ensure data consistency and correct execution order. The need for synchronization arises due to data dependencies, shared memory usage, and the sequential nature of kernel launches.

This is why identifying synchronization points is essential to ensure data consistency and correct execution order. Preventing race conditions and ensuring the correctness of the DWT process.

Within 1D DWT Kernel - After each stage of computation within the kernel, a synchronization point is required to ensure all threads have completed their convolution and downsampling operations before proceeding. This can be achieved by synchronizing the threads with the respective fucntion.

Between 1D DWT Kernels - After launching the 1D DWT for rows, columns, and depth slices, synchronization is required to ensure the 1D DWT on rows is completed before starting the 1D DWT on columns, and similarly for depth slices. If this is not done, then the data will be corrupted as the next DWT will be applied to the data before the previous DWT has been applied. This synchronization point will essentially have to slow down the process with a theoretical wait but is a necessary step coming from the algorithms' sequential nature.

Multi-Level DWT - Between levels in the multi-level function, synchronization is required to ensure the DWT for the current level is completed before resizing the volume and starting the next level. This is important as the data from the previous level is required to be processed in the next level. Once again this comes from the sequential nature of the algorithm and the need to process the data in a specific order.

Both the multi-level synchronization and the between 1D DWT kernels synchronization can be handled by synchronizing the cuda threaded devices with the respective function.

2.3 Secquenced Plan for CUDA Implementation

Following the above analysis, we can outline a sequenced plan for the CUDA implementation of the 3D DWT process:

Handling I/O Operations - The I/O operations, such as loading and saving the 3D volume data, will be handled by CUDA functions that read and write data from/to binary files. The loading in particular will handle the data to be loaded into the GPU's global memory. This set's up the data for parallel processing using CUDA and involves allocating memory on the GPU and copying the data from the host to the device.

Performing 1D DWT on Rows, Columns, and Depth Slices - The 1D DWT will be applied to the rows, columns, and depth slices of the 3D volume concurrently using CUDA kernels. Each kernel will process a row, column, or depth slice of the volume, applying the 1D DWT using wavelet filters. Synchronization within the kernel will be achieved to ensure correct execution order.

Multi-Level DWT - The multi-level DWT function will be implemented using CUDA, applying the 3D DWT multiple times to achieve multi-level decomposition. Handling the halving of the dimensions between levels and synchronization points to ensure data consistency and correct execution order.

Selecting Block and Thread Sizes - Once a stable implementation is achieved, the number of threads per block and blocks in the grid will be manually tested to find the optimal values for the particular GPU setup. This will involve testing different configurations to maximize GPU utilization while avoiding overfitting to the GPU's capabilities.

By mapping these functions to the CUDA architecture, we can leverage the parallel processing capabilities of the GPU to significantly improve the performance of the DWT process on a 3D volume. Synchronization points are carefully placed to ensure data consistency and correct execution order, preventing race conditions and ensuring the correctness of the DWT process.