High Performance Computing Image Analysis for Radiotherapy Planning

Zilong Pan

August 20, 2013

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2013

Abstract

The Edinburgh Cancer Centre at the Western General Hospital in Edinburgh is doing research on image analysis for predicting lung fibrosis induced by radiation as part of a treatment plan. They are developing a MATLAB code to analyse three dimensional Computed tomography (CT) images of patients but, because a standard three dimensional CT image is a large data set to be processed, the original MATLAB code runs very slowly and takes a long time to produce a result.

This project tackles the challenge of processing large data sets of three dimensional CT images and accelerating the original MATLAB code. The project focuses on the most computational demanding part of the code, which is filtering the three dimensional CT images with a Gabor filter. We improve the image filtering algorithm with a Fast Fourier Transform, and apply multi-cores and GPU techniques for parallelisation. In this project, three GPU technologies are used to optimise the original MATLAB code. The first one is the MATLAB Parallel Computing Toolbox. The second is AccelerEyes' Jacket,[1] which is a GPU based accelerator for MATLAB. The third is CUDA from the NVIDIA Company, being hybrid programming with MATLAB through MEX files, which are MATLAB Executable files. Combining MATLAB with GPU techniques and algorithm improvements, the original MATLAB code is optimised largely, and has achieved a speedup of 120.

[1] Jacket is no longer produced as it was bought by MathWorks who plan to incorporate the functionality in a future release of MATLAB.

Contents

List of Tables

List of Figures

Acknowledgements

Firstly I offer my sincerest gratitude to my supervisors: Dr Mario Antonioletti and Mr Luis Cebamanos from EPCC and Dr Bill Nailon from the Department of Oncology Physics at the Western General Hospital. They have provided me a lot of guidance with knowledge and patience and have given me sufficient materials for the project, and have helped me solve problems in the project as they have arisen.

I would like to offer my gratitude to Dean Montgomery, who provided me the original code and helped me understand the code.

I would like to thank Qian Du for her encouragements and suggestions.

I would like to thank my parents for supporting me during my study in the UK.

Chapter 1

Introduction

Radiotherapy has been one of the most effective approaches for cancer treatment for several decades [1]. It uses radiation beams to try to kill tumours inside the bodies of patients, but the radiation beams used may also hurt the surrounding tissue as well [1]. For lung cancer treatment, radiotherapy has a 13% to 37% possibility of causing lung fibrosis [1]. The onset of lung fibrosis is currently not visible by eyeball inspection of Computed Tomography (CT) scans in the sense that we do not as yet know what to look for, but can be classified through Texture analysis [2]. According to a significant amount of research, texture analysis can be used to identify the regions of a lung that have already changed due to lung fibrosis [1]. The difference between healthy lung tissue and interstitial lung diseases on CT images can be classified through volumetric texture features.

The Edinburgh Cancer Centre at the Western General Hospital in Edinburgh is doing research in image analysis algorithms, and is developing a MATLAB application for three-dimensional CT image analysis to predict lung fibrosis induced by radiation. The three-dimensional texture code calculates texture features from a three-dimensional CT image which are then used to train a machine learning algorithm which once trained is reasonably successful at predicting the onset of radiation fibrosis. However, the original MATLAB code which does the image feature extraction is very time-consuming and takes a long time to analyse a three-dimensional CT image. For example, it takes more than 5 days to analyse a volume of 512 * 512 * 300 pixels on Fermi0. It leads to large inconvenience for the lung cancer research. This is a good chance to accelerate a program, and compare different kinds of optimisation approaches.

This project aims to accelerate the original MATLAB code using different HPC approaches, and focuses on how to process 3D image data in a faster way. In a previous MSc project, a two-dimensional visual texture analysis MATLAB code was parallelised using different methods [6], and now the visual texture analysis code is being extended to three dimensions, which is much more computationally demanding than the two-dimensional code. In order to tackle the challenge, we use the Fast Fourier Transform (FFT) based convolution to improve the image filtering algorithm, and we also parallelise and accelerate the MATLAB code using Graphics Processing Unit (GPU) techniques.

The development environment used is MATLAB, as the Edinburgh Cancer Centre use MATLAB for their research and development. MATLAB provides a large number of built-in functions for matrix calculation and image processing, and it can be combined with GPU techniques for parallelisation. In this project, we have done several improvements to the original MATLAB code, including:

- Changing the three dimensional image filtering to use a Fast Fourier Transform (FFT) based convolution algorithm instead of the MATLAB built-in function `imfilter`. The FFT based convolution alone achieved a 27 time improvement in speed. We further improved the transform from convolution to correlation, which is using conjugation approaches instead of rotating filter arrays, which benefits the memory, and reaches a speedup of 57 times over the original algorithm for 64 * 64 * 37 test case, and 99 times for 512 * 512 * 300 test case.

- We parallelised the original code to use multiple CPU cores using the MATLAB Parallel Computing Toolbox. Based on the original algorithm, the parallelised code can get 8 times speedup when it is run on 12 cores. The performance results are used to analyse the speedup with different number of cores, and also, the performance of CPU code is compared with the performance of GPU code in later stages.

- We accelerated the original code with GPUs using the MATLAB Parallel Computing Toolbox (PCT). The PCT provides some supports for GPU computation. It can transfer data from the MATLAB working space to a GPU, and provides some built-in functions that support GPU computation. However, some of the built-in functions do not support three-dimensional arrays for GPU even in the latest version of MATLAB (R2013a), such as the function of "imfilter" [10]. This GPU code has more than 100 times speedup on both Fermi0 and Comp002 over the original code.

- A product of AccelerEyes named Jacket is used to parallelise the original MATLAB code with GPUs. Jacket is a GPU accelerator add-on for MATLAB [13]. It provides better supports and improvement for GPU computation than the Parallel Computing Toolbox. In addition, it provides a much easier way for GPU programming rather than through using CUDA directly. Its GPU code has 120 times speedup over the original code. However, Jacket is no longer sold to customers as its functionality will be integrated into the future version of MATLAB. Jacket is still available in the MATLAB R2012a on Fermi0.

- Fifthly, we accelerate the original MATLAB code with GPUs using CUDA/C. CUDA is one of the most mature GPU programming models [17], and is developed by the NVIDIA Company. It can achieve the highest performance on the NVIDIA GPU cards [17]. Furthermore, CUDA has a professionally optimised FFT library for GPU computation named CUFFT [22], which gives great performance and convenience. In addition, CUDA code can be called from MATLAB using the MEX-files, which are the MATLAB Executable files. MEX files can call other languages, such as CUDA.

In the remainder of this dissertation Chapter 2 describes the background theory for the image analysis used, such as texture analysis, Gabor filter, FFT based convolution and so on. Chapter 3 presents the profiling, design and implementation of the optimisation to the original code, including algorithm improvement, multiple CPU-cores parallelisation, and GPU acceleration. Chapter 4 illustrates and analyses the results and speedup, and compares the performance among different approaches, different testing sizes and different machines. Chapter 5 discusses a conclusion and further work.

Chapter 2

Background Theory

2.1 Radiotherapy

Radiotherapy has been playing a significant role in cancer treatment for decades [1]. It applies high energy radiation beams, which pass through a human body to try to kill cancerous tumours inside the patient's body [1, 2]. However, the radiation beams will also possibly damage the surrounding tissue which is not cancerous. According to research on lung cancer treatment by the Western General Hospital, after radiotherapy there is a 13%-37% possibility of the patient suffering from radiation induced lung fibrosis when lung tissue has been irradiated [1]. The over exposure of healthy lung tissue to radiation beams causes abnormal changes to the lung tissue, and will lead to lung fibrosis, which is a very serious lung disease which permanently affects the oxygen transfer in the lung [1]. Therefore, it is crucial to minimise the radiation hurts by recognising the regions of cancerous tumours and predicting the risk of side-effects as accurately as possible.

Figure 2.1 shows a photo of the linear accelerator (LINAC) radiotherapy device at the Western General Hospital in Edinburgh.

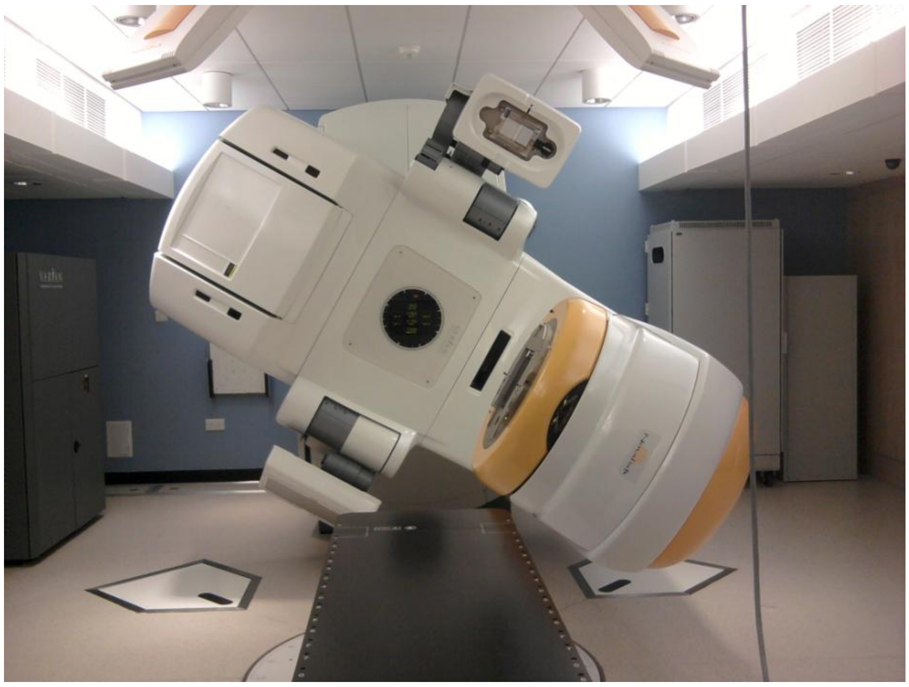


Figure 2.1: The radiotherapy device LINAC at the Western General Hospital in Edinburgh

2.2 Three Dimensional Texture Analysis

The Edinburgh Cancer Centre at the Western General Hospital uses three dimensional CT images for texture analysis. Three dimensional CT images can give more anatomic details than two dimensional images. The orientation of different slice views on a 3D image are shown in Figure 2.2, and Figure 2.3 shows the corresponding lung slices (actual MATLAB data) from 3 different views from a 3D CT image.

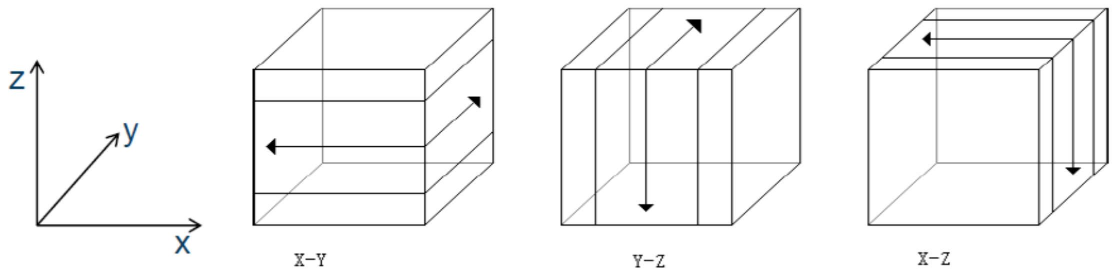


Figure 2.2: Different views of slices on a 3D image.

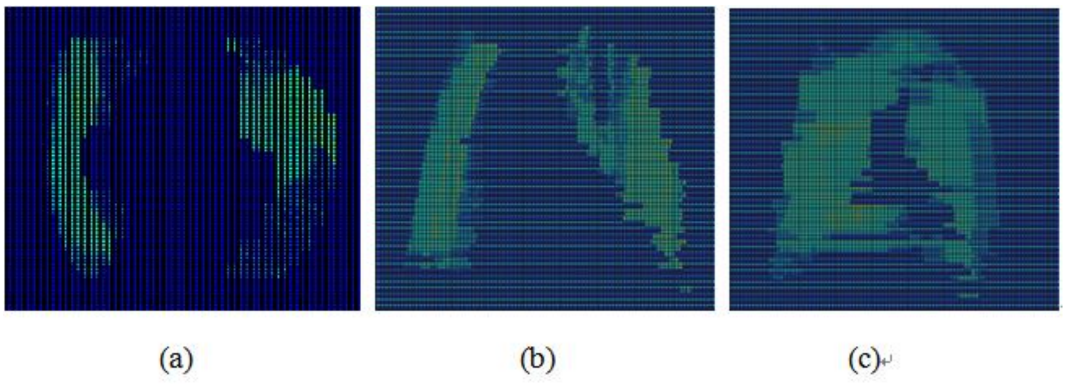


Figure 2.3: The CT slices of a lung (actual MATLAB data) from 3 different views:

(a) X-Y plane. (b) Y-Z plane. (c) X-Z plane

Although lung fibrosis is often not visible from CT scans, it can be classified by texture analysis [1, 4]. CT Image analysis techniques have been applied for the classification of lung fibrosis. The method can identify the areas of a lung that have some changes due to lung fibrosis. Figure 2.4(A) shows a CT image of a lung that has no side-effects after radiotherapy, and (B) is a CT image of a lung which suffers fibrosis.

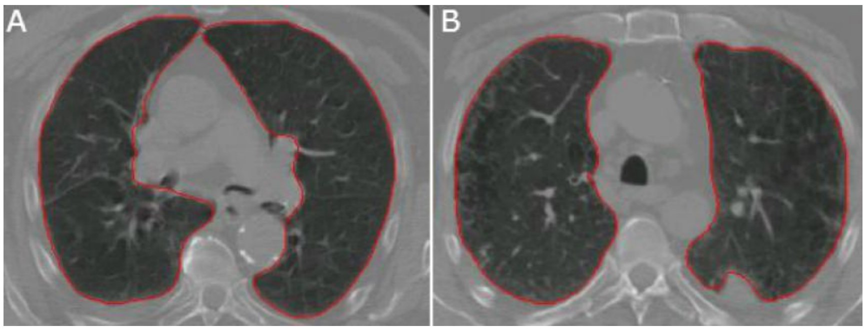


Figure 2.4 [1]: lung CT images after radiotherapy: (A) non-fibrosis, (B) fibrosis.

The Western General Hospital has done research on predicting the risk of fibrosis by calculating the visual texture properties through 3D CT scans. Their texture analysis MATLAB code can classify the surrounding tissue of a lung, and calculates a total number of 2139 texture features on a three-dimensional CT image a simplified set of which is then used train a machine learning algorithm which is then used to predict whether a patient is likely to develop radiation induced fibrosis from the CT images of their lung. The image features are based on the features calculated from the algorithms described below.

2.2.1 First Order Statistics (FOS)

The First Order Statistics calculate seven texture features from a grey level image which are: the mean, variance, coarseness, skew, kurtosis, energy and entropy [1]. They are the basic texture features of an image. The calculation of the FOS is only based on individual pixels, and has no computational relationship with other pixels.

2.2.2 Second Order Statistics (SOS) and Haralick Features

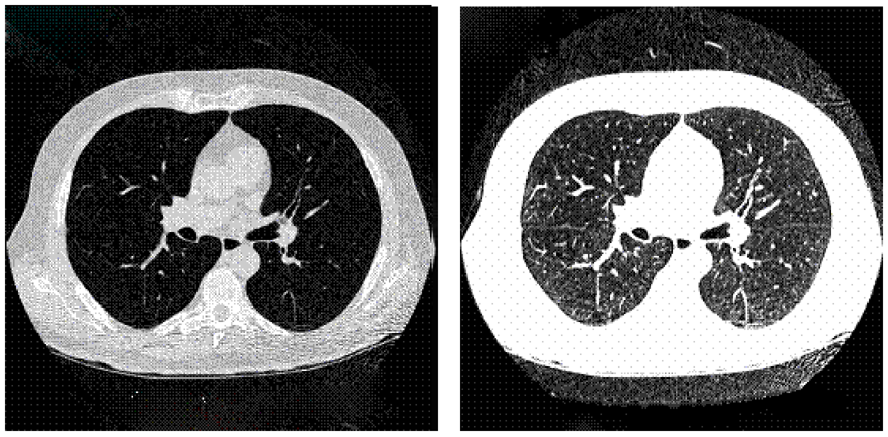
The Second Order Statistics are an extension of FOS, and calculate the features of pairs of pixels which are named Haralick features [1]. The pair pixels are chose according to the probabilities that a pixel will be found adjacent to another pixel on a 3D grey level image. The calculation of the Haralick texture features considers the adjacency between two pixels. The Second Order Statistics are calculated with the co-occurrence matrices which will be described in the next section.

2.2.3 Grey Level Co-occurrence Matrix (GLCM)

The Grey Level Co-occurrence Matrix (GLCM) is used to calculate the Second Order Statistics and the Haralick features [1]. The elements of the GLCM are the probabilities of becoming a pair-pixel between two points in a 3D grey level image. The texture analysis MATLAB code generates 13 GLCMs describing the relationship between 2 points in all directions for a 3D grey level image. The GLCMs are the used to calculate of the Second Order Statistics and Haralick features.

2.2.4 Gabor Filtering

Gabor features are calculated by image filtering with the Gabor filter named after Dennis Gabor [1, 3]. A Gabor filter is a linear filter, and is particularly suitable for edge detection and texture representation, because it is able to simulate the human visual system [6]. Two dimensional Gabor filtering has been widely used and achieved great success in image analysis and processing, for example, texture segmentation, face recognition and so on [7]. Figure 2.5 shows the enhanced effect of using Gabor filtering to a lung CT image.



(a) Original image (b) Image by Gabor filtering

Figure 2.5: A lung CT image is enhanced by Gabor filter [7].

On the topic of lung texture analysis, the Edinburgh Cancer Centre has expended Gabor filter into 3D. The 3D Gabor filter is using 36 angles and 4 spatial frequencies to filter a 3D CT image. There are 144 texture features calculated by the three dimensional Gabor kernel which is modulated by a sinusoidal wave in the original 3D MATLAB code.

The 3D Gabor kernel is defined as shown below [1]:

$$\varphi_{f,\theta,\phi} = S \times \exp(-((\frac{x'}{\sigma_x})^2 + (\frac{y'}{\sigma_y})^2 + (\frac{z'}{\sigma_z})^2)) \times \exp(i2\pi(xu + yv + zw))$$

Where u = Fsinφ cosθ , v = Fsinφ sinθ , w = Fcosφ , F = sqrt((u2 + v2 + w2)^2), x’ = x cosθ - ysinθ , y’ = cosφ (xsinθ + ycosθ) - zsinφ , z’ = sinφ (xsinθ + ycosθ) + zcosφ . The S is a normalisation scale, and σ is the width of the Gaussian envelope. The (u, v, w) are the frequency of the sinusoid wave, and F is the amplitude of the wave. The φ and θ are the orientations of the wave in the three-dimensional frequency domain. Using different parameters can generate different Gabor filters, which are used for filtering images to get texture features.

2.3 Convolution and Cross-Correlation

The original MATLAB code used a MATLAB built-in function `imfilter` to do the linear filtering, which has two modes including correlation and convolution [10].

2.3.1 Convolution

Convolution is defined as follows [7]:

$$g * h = \sum_{x,y,z}\sum_{i,j,k} g(x,y,z)h(x-i, y-j, z-k) ,$$

Where g represents the 3D image data and h is the 3D filter to be applied. The g(x, y, z) is a point on the 3D image, and h(x-i, y-j, z-k) is the filter weights corresponding to the point of g(x, y, z) on the image. The progress of doing linear filtering using convolution is as follows:

1) Zeros are added to the halo of the image, and the width of the halo is half of the filter size. So the size of image with padding zeros is: size (image) + size (filter) – 1.
2) The filter is moved onto the image, and the centre of the filter is made to correspond with a point on the image. The filter will cover a region of the image.
3) On the covering region, each point of the image is multiplied by the responding weight of the filter, and then the products are summed up as the result of the central point.
4) The filter is moved to the next point in the image, and the process is repeated using a loop from the second step until one finishes operating on all the image points.

For example, taking a 2D convolution, Figure 2.6 shows a convolution between a 5x5 image and a 3x3 filter. The result of the point (2,4) is 1x2 + 8x9 + 15x4 + 7x7 + 14x5 + 16x3 + 13x6 + 20x1 + 22x8 = 575.

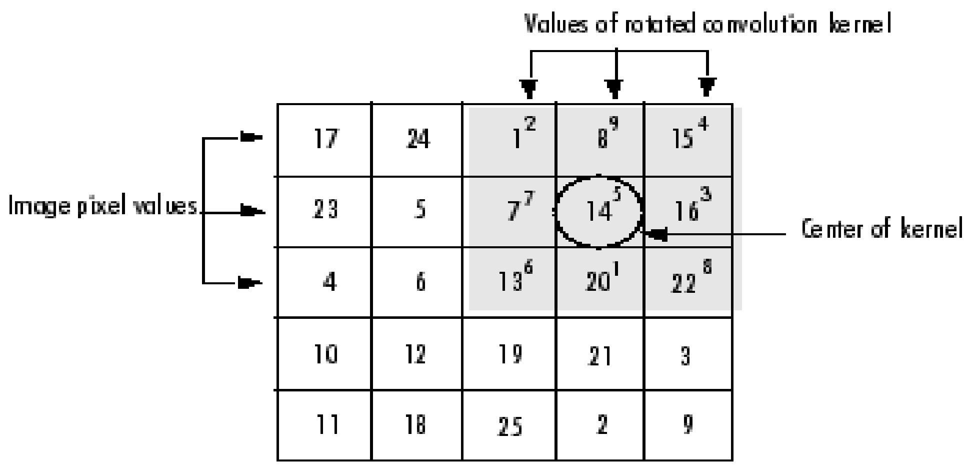


Figure 2.6: An example of 2D convolution.

2.3.2 Cross-Correlation

The `imfilter` function in the original MATLAB code actually does cross-correlation between the image and the filter. Cross-correlation is defined as follows [7]:

$$g \otimes h = \sum_{x,y,z} \sum_{i,j,k} g(x,y,z) h(x+i, y+j, z+k)$$

,

Where g represents the 3D image data and h is the 3D filter to be applied. The g(x, y, z) is a point of the 3D image, and h(x+i, y+j, z+k) is the filter weights corresponding to the point of g(x, y, z).

Cross-correlation is actually equivalent to convolution with the filter rotated 180 degrees [12]. Thus, doing linear filtering using cross-correlation is nearly the same as convolution, and the algorithm is described below:

1) The 3D filter is rotated with 180 degrees firstly, which is filter = filter (end: -1 :1, end: -1: 1, end: -1: 1) in MATLAB syntax.
2) Do the convolution as in the description above and get the results.

For both the convolution and the cross-correlation, the size of the calculation results is size (results) = size (image) + size (filter) – 1. We extract the central part of the results that is the same size as the image, and ignore the halos.

For the original MATLAB code, the `imfilter` is doing cross-correlation. If we use convolution, we have to rotate the filter first, and then do calculation. If we use cross-correlation, we can do calculation directly.

2.4 FFT based Convolution and Cross-Correlation

The original implementation of the MATLAB code used the `imfilter` function contained in the Image Processing Toolbox – one of the MATLAB add-ons. What the `imfilter` actually does is a convolution or cross-correlation between the image and the filter. It costs more to do the linear filtering directly with large data sets, and fortunately, there are algorithms that allow convolutions or cross-correlations to be done more quickly instead. The most commonly used fast image filtering algorithm is based on the Fast Fourier Transform (FFT) [7].

FFT is one of the most significant numerical algorithms in image processing, and is widely used in many areas, such as image analysis, signal processing, and so on. The FFT is able to compute the discrete Fourier Transform in a very fast way, and it converts space or time to frequency, or inverse from frequency to space[7].

2.4.1 FFT based Convolution

The FFT based convolution applies the property of the Convolution Theorem, which is defined as follows [7]:

convolution (image, filter) = IFFT (FFT(image) .* FFT(filter)) ,

where IFFT means inverse FFT, and the ".*" means pointwise product.

The convolution between an image and a filter can be calculated by taking FFTs of the image and the filter respectively and then the convolution result can be calculated by performing an inverse FFT on the point-wise product of the two previous FFTs of the image and filter.

Because the FFT wave is periodic, we need to expand the size of the FFT period to be larger than the convolution size, which is equal to size (image) + size (filter) - 1. In other words, the period of FFT should be larger than: size (image) + size (filter) – 1. The image and the filter should be padded with zeros. According to the FFTW, FFT gets the highest efficiency if the size of the FFT equals a power of 2. Finally, the results are extracted from the central part of the whole results.

2.4.2 FFT based Cross-Correlation

The FFT based Cross-Correlation applies a property that is analogous to the convolution theorem, and the mathematical theory is defined as follows [7]:

correlation (image, filter) = IFFT (conjugation (FFT(image)) .* FFT(filter)).

The cross-correlation between an image and a filter is similar to the convolution, and it can be calculated by taking FFTs as well. After doing FFTs to the image and the filter respectively, we do conjugation to the FFT of the image. Here we use conjugation instead of rotating the filter, so that it can get higher efficiency in computation. Finally, an inverse FFT is performed to the point-wise product of the conjugated FFT image and the FFT filter.

Similarly to the FFT based convolution, the FFT based cross-correlation requires the period of FFT to be larger than size (image) + size (filter) – 1. The image and the filter should be padded with zeros before performing the FFTs. In the end, the desired results are extracted from the central part of the whole result.

2.4.3 Complexity Comparison

Assuming that the image size is m * m * m, and the filter size is n * n * n (n<<m), then the calculation complexity of the direct image filtering is $O(m^3n^3)$, while the FFT based image filtering is $O((m+n)^3 \log(m+n)^3) = O((m+n)^3 \log(m+n))$. From the comparison, we can see that the FFT based algorithm can reduce the calculation complexity significantly, but the actual comparison still depends on the size of the coefficient in front of each of the order calculations.

2.5 General Purpose on Graphics Processing Unit (GPGPU)

2.5.1 GPU Computing

The Graphics Processing Unit (GPU) for certain types of problem can achieve high performance in floating point calculation, and has thus been applied to high performance computing problems to reach large speedup [17]. The GPU acts as an accelerator to the CPU. The CPU executes most of the lines of code, while the GPU runs the key computational kernels, which can take advantages of large number of cores and high graphics memory bandwidth.

GPGPU (General Purpose computation on GPUs) has become popular in scientific computing [18]. Figure 2.7 shows the development in peak performance of GPUs and CPUs.

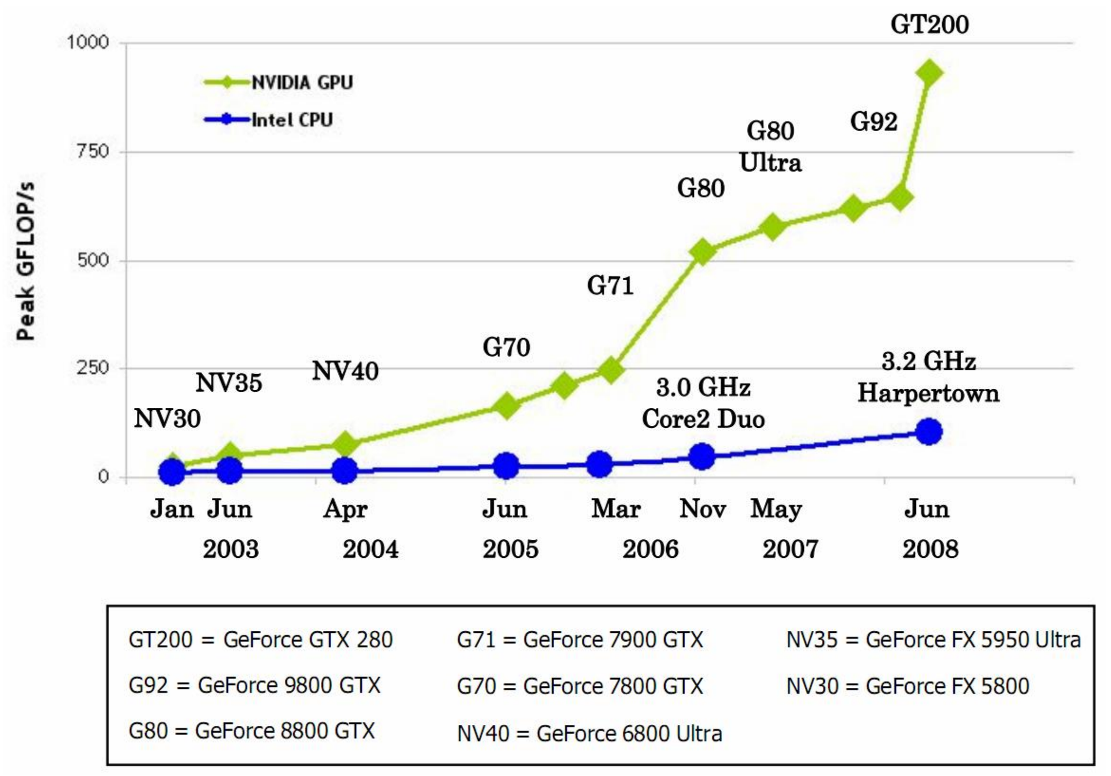


Figure 2.7 [7]: The development of GPUs and CPUs from 2003 to 2008

We have to consider the data transfer between the CPU and the GPU. If the data transfer costs more time than calculation, it is no need to use GPU, but for certain types of computational intensive applications, we can use a GPU together with a CPU to accelerate computing.

GPU computing provides helps for acceleration by offloading the computationally intensive part of a code to the GPU for large amount of parallel computing, while the other part of the code still runs on CPUs. The combination of CPUs and GPUs is becoming more and more popular and powerful for some kinds of computational intensive programs.

2.5.2 GPU Programming

The GPU programming models we use in this project are the GPU support included in the MATLAB Parallel Computing Toolbox, the Jacket add-on for MATLAB, and CUDA/C called from within MATLAB. Each of these is discussed in more detail below.

The Parallel Computing Toolbox (PCT)

MATLAB provides its PCT for parallel computing, and it is simple for programmers to use for parallelising and accelerating a serial code [9]. The PCT not only supports parallelisation with multiple CPU cores, but also provides supports for GPU acceleration [11].

The PCT provides 171 built-in functions for GPU computing currently [12], including its FFT functions, which have already been highly optimised, and perform very well. The FFT functions in MATLAB are based on the FFTW and CUFFT [12], which have the very top performance in the FFT implementation on CPUs and GPUs respectively.

In addition, the MathWorks Company has realised the power and importance of GPU computing, and they bought the GPU add-on Jacket from the AccelerEyes Company in December 2012 [13]. The PCT is integrating the highly optimised GPU add-on Jacket into the PCT. The PCT will have more supports and higher performance on GPU computing in the near future. Jacket is available on Fermi0 as a separate add-on in MATLAB 2012a, but it is not installed on Indy.

Jacket

Jacket is a GPU computing add-on produced by the AccelerEyes Company [13], and it has great supports for GPU computing. Jacket has provided as many as 589 GPU functions, compared with 171 in MATLAB 2012 [14]. Jacket not only provides a wider range of GPU functions, but also provides better acceleration with GPUs.

According to a number of benchmark reports, Jacket's GPU functions have higher performance than the PCT, and much easier for GPU programming than CUDA or OpenCL.

As mentioned above, Jacket has been purchased by the Mathworks [13], and is being integrated into MATLAB Parallel Computing Toolbox, but it is still available currently to existing customers – no new purchases can be made though.

CUDA

The Compute Unified Device Architecture (CUDA) is a mature GPU programming model developed by the NVIDIA Company [17]. CUDA has the highest performance and flexibility for GPU programming on NVIDIA GPU cards, but it is much more difficult to program and requires much more efforts than using the PCT or Jacket. We have to consider the data transfer between the host and device, 3D data memory access, number of threads, and many other relevant programming details.

CUDA has its own FFT library named CUFFT [22]. The CUFFT library is modelled after the FFTW, which is one of the most efficient CPU implementation of FFT [22]. The FFT is a divide-and-conquer algorithm for efficiently calculating discrete Fourier transform. The CUFFT provides a simple interface for GPU computing of FFT.

In this Chapter, we describe the background theory for the image analysis used, such as texture analysis, Gabor filter, FFT based convolution, and GPU computing and programming. We will present the profiling, design and implementation of the optimisation to the original code, including algorithm improvement, multiple CPU-cores parallelisation, and GPU acceleration.

Chapter 3

Design and Implementation

In order to optimise the original 3D MATLAB code, we first analysed and profiled the code using the MATLAB Profiler, which indicates where the expensive parts of the original code are. After the profiling, the time-consuming part of the code was accelerated with different methods, including:

- Algorithm improvement
- Using multiple CPU cores with MATLAB Parallel Computing Toolbox (PCT)
- Using a GPU with the MATLAB PCT
- Using a GPU with Jacket
- Using a GPU with CUDA/C

These topics are discussed in the remainder of this chapter.

All of the implementations are based on the MATLAB environment. MATLAB has advantages in scientific programming, and provides a number of functions for matrix calculation and image processing. Besides that, MATLAB supports GPU computing and calling CUDA, which gives us convenience for optimisation. In addition, the Edinburgh Cancer Centre can continue their research and development on the optimised MATLAB code.

3.1 Analysis of the Original Code

The original 3D MATLAB code generates a 3D texture feature vector which is subsequently used in an analysis for predicting whether lung fibrosis will result in a patient from their CT images. The working procedure for generating the feature vector is schematically shown in the Figure 3.1. When a volume CT image is input, the application transforms the CT image to a grey level image, and generates different matrices which are used to calculate relevant texture features in the next stage. The calculation of 3D texture features includes First Order Statistics, Second Order Statistics, Haralick features, Gabor features and so on. Most of which have already been described in Chapter 2. The results of the texture features are output and used in subsequent analysis to predict whether there is a chance of developing lung fibrosis but this is out of scope for this project.

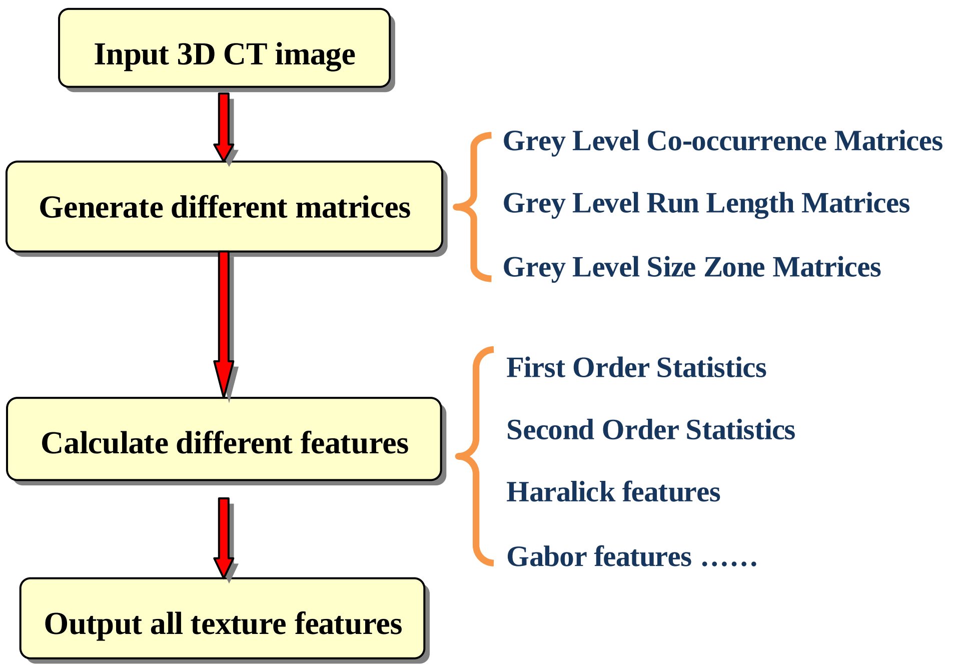

Input 3D CT image
Generate different matrices
Grey Level Co-occurrence Matrices
Grey Level Run Length Matrices
Grey Level Size Zone Matrices
Calculate different features
First Order Statistics
Second Order Statistics
Haralick features
Gabor features ……
Output all texture features

Figure 3.1: The procedure diagram of the original texture analysis code

The original MATLAB code is mainly constructed of 8 parts:

- The main function *calc_3d_stats* gets the input volume image, and calls the other functions to complete the jobs of generating matrices and computing texture features. Finally, it put all the texture features into a vector as the output result.
- The *graycomatrix* and *graycomatrix3Ddm448* functions generate the gray-level co-occurrence matrix, which is used for the calculation of the Haralick Features and the Second Order Statistics.
- The *glrlm_DICOM* function creates the Grey-Level Run Length Matrix, which is used for the Second Order Statistic and Higher Order Statistics.
- The *fos_3d* function implements the First Order Statistics for three-dimensional volumes.
- The *Haralick_features* function computes the Haralick texture features from the co-occurrence matrices.
- The *gabor_3d_features* function applies Gabor filters to calculate energies.
- The *glszm_3d* function calculates texture features from Zone Matrices.

- The *grayrlprops* function calculates run length statistics from the run length matrices.

Figure 3.2 shows the main structure of the texture analysis code.

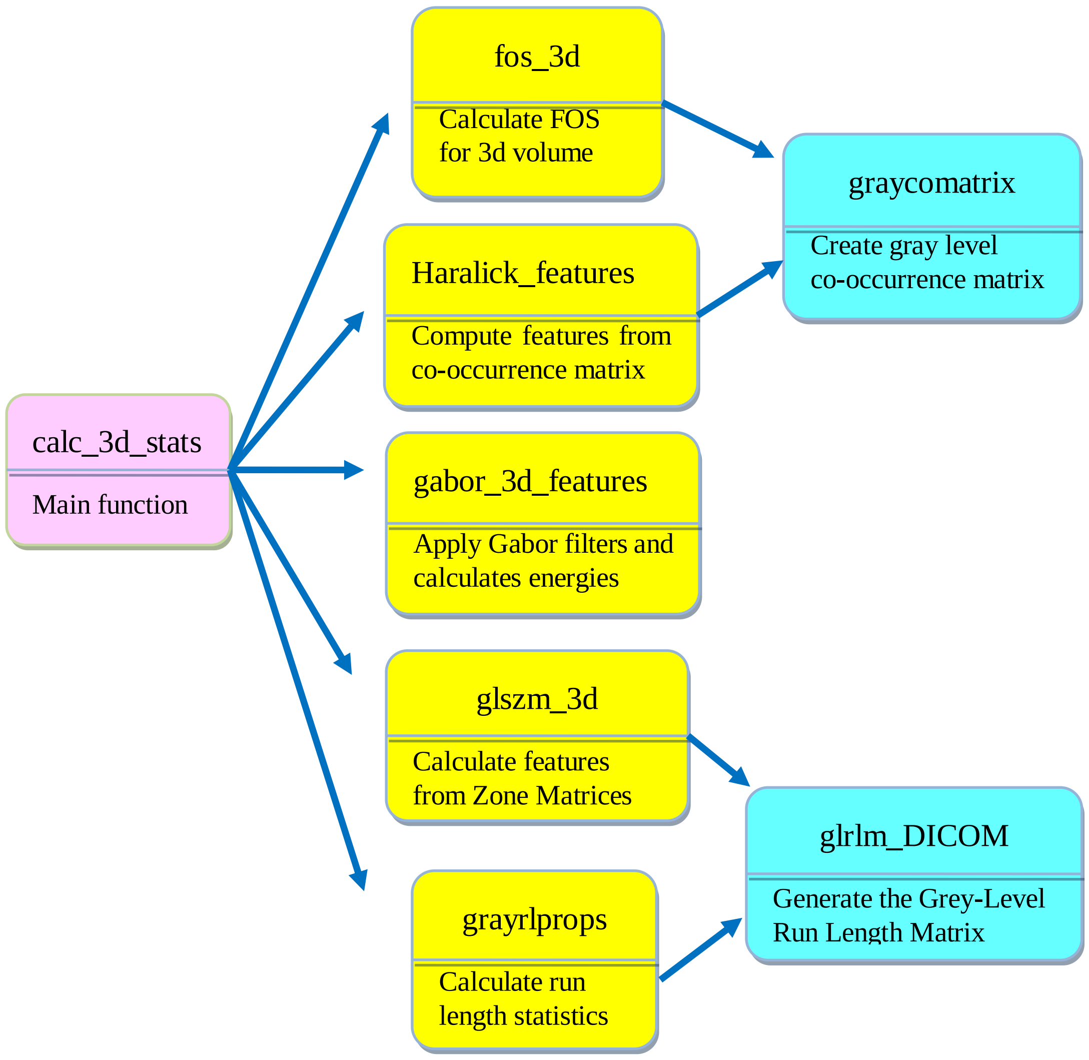

fos_3d
Calculate FOS for 3d volume
graycomatrix
Create gray level co-occurrence matrix
Haralick_features
Compute features from co-occurrence matrix
calc_3d_stats
Main function
gabor_3d_features
Apply Gabor filters and calculates energies
glszm_3d
Calculate features from Zone Matrices
glrlm_DICOM
Generate the Grey-Level Run Length Matrix
grayrlprops
Calculate run length statistics

Figure 3.2: The main structure of the texture analysis code

3.2 Profiling the Original Code

In order to find out where the code spends much time, the original MATLAB was tested and profiled. We used MATLAB Profiler at first, and then set timers (*tic* and *toc*) in the code to get more accurate elapsed time of different parts.

The Edinburgh Cancer Centre has provided 4 test cases. All of them are three dimensional CT images, and their sizes are respectively:

1. 64 * 64 * 37 pixels,
2. 128 * 128 * 75 pixels,

3. 256 * 256 * 150 pixels, and
4. 512 * 512 * 300 pixels.

The testing was taken on MATLAB (R2012a) on the backend node Fermi0 of Hydra. Fermi0 has 4 Intel Xeon X5650 (2.67GHz) CPUs with 6 cores on each processor. Fermi0 also hosts 4 GPUs of NVIDIA Tesla C2050.

Using the original MATLAB code all of the four test cases took a very long time to run. The timing results are shown in Figure 3.3. The smallest test case (64 * 64 * 37) spent 776 seconds (about 13 minutes). The 128 * 128 * 75 image ran for 1.8 hours. The 256 * 256 * 150 pixel image took nearly 15 hours to process. For the largest test case (512 * 512 * 300), which is a standard 3D CT image size, it took 425011 seconds -in other words, it spent about 5 days to process a standard 3D CT image.

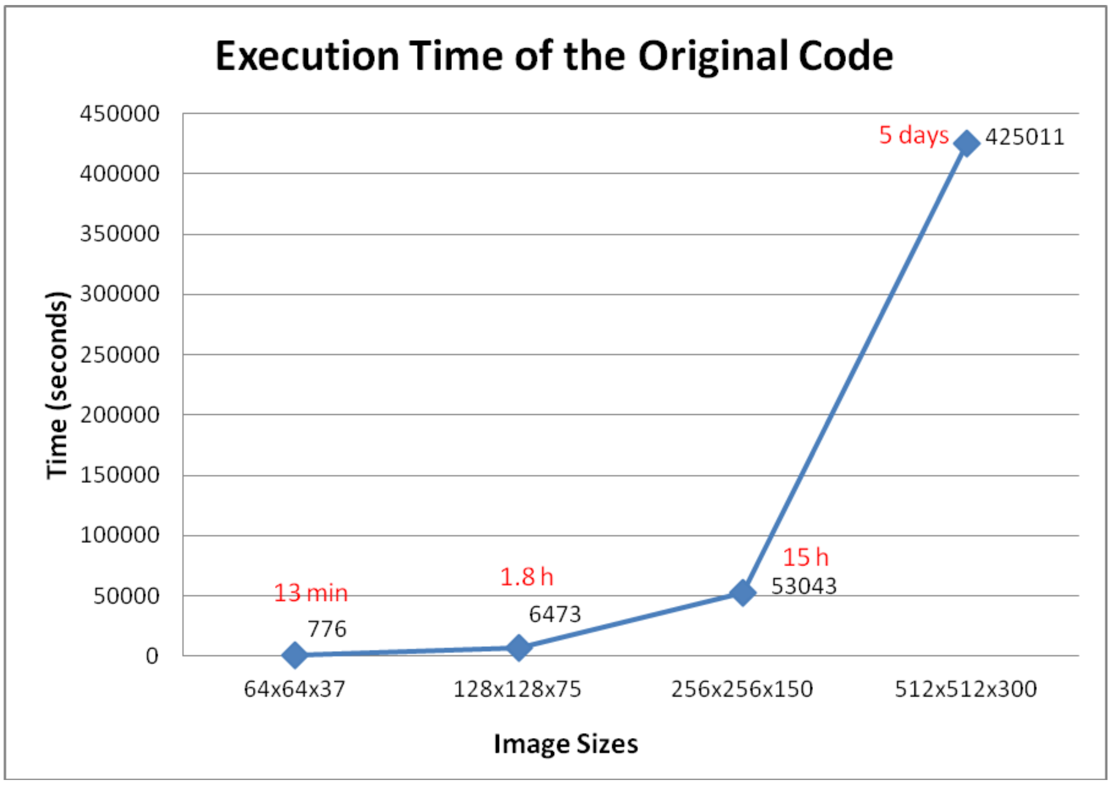
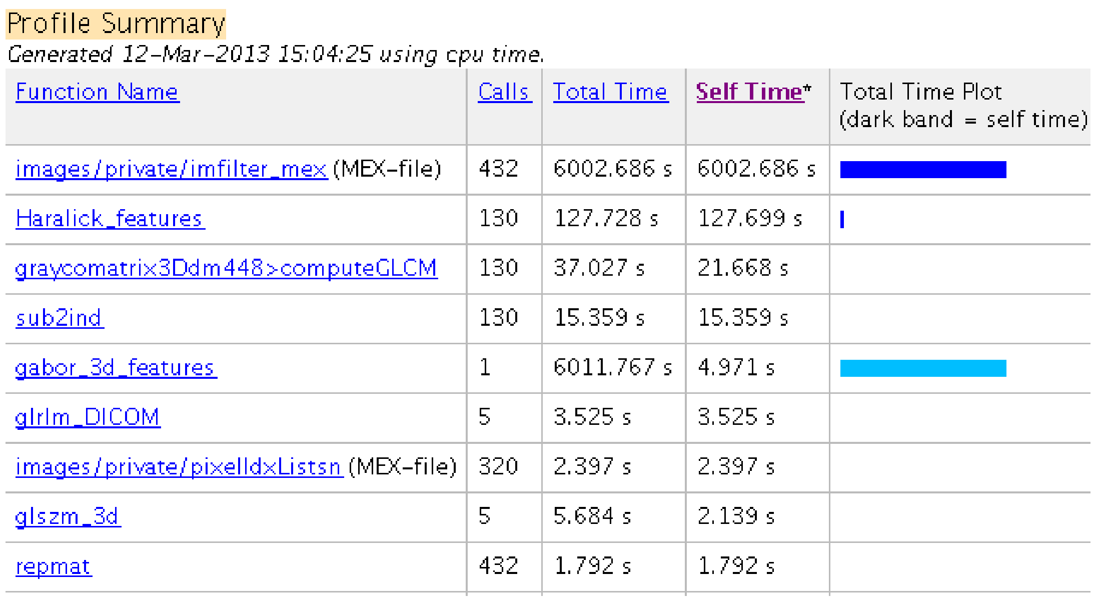


Figure 3.3: Timing results of 4 test cases for the original code

However, it is necessary to profile the original MATLAB code in order to find out where the time-consuming parts are before optimising the code.

At first, the code was profiled using the MATLAB Profiler. All the profiling results from the 4 test cases clearly pointed out that the `imfilter` routine in the `gabor_3d_features` function took up almost the whole execution time. It accounted for more than 99% of the total run time. Therefore, the project focused on the Gabor filtering, which would benefit the most from being optimised and parallelised.

One of the profiling results is given out in Figure 3.4 as an example (128 * 128 * 75). The MEX-file is the MATLAB Executable file.


Profile Summary

Generated 12-Mar-2013 15:04:25 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
images/private/imfilter_mex (MEX-file)	432	6002.686 s	6002.686 s	▬
Haralick_features	130	127.728 s	127.699 s	▏
graycomatrix3Ddm448>computeGLCM	130	37.027 s	21.668 s	
sub2ind	130	15.359 s	15.359 s	
gabor_3d_features	1	6011.767 s	4.971 s	▬
glrlm_DICOM	5	3.525 s	3.525 s	
images/private/pixelIdxListsn (MEX-file)	320	2.397 s	2.397 s	
glszm_3d	5	5.684 s	2.139 s	
repmat	432	1.792 s	1.792 s	

Figure 3.4: Profile Summary for 128x128x75 image

In order to get more accurate elapsed time, we set timers (*tic* and *toc*) in the part of Gabor filtering instead of using the MATLAB Profiler, which would take up some execution time itself. The more accurate timing results are shown in Table 3.1.

Size	***Total time(sec)***	***Gabor time(sec)***	***Proportion (%)***
64 * 64 * 37	776	771	99.4%
128 * 128 * 75	6473	6450	99.6%
256 * 256 * 150	53043	52846	99.6%
512 * 512 * 300	425011	423470	99.6%

Table 3.1: The timing results and proportion of the Gabor filtering

It is clear that the part of Gabor filtering accounts for more than 99%, almost the total execution time of the original MATLAB code. Therefore, the project focused on the part of Gabor filtering for optimisation, which is presented in later sections.

3.3 Algorithm improvement

3.3.1 imfilter

The original MATLAB code uses the `imfilter` function from the Image Processing Toolbox in MATLAB for 3D image filtering [10]. The `imfilter` function in the code uses 3D Gabor kernels to filter the input 3D CT image, in order to calculate the Gabor texture features. The `imfilter` has two possible working modes: cross-correlation and convolution as mentioned in Chapter 2.1, which uses cross-correlation by default. So the line of code:

```
    imfilter (image, filter, ‘same’)
```

does a cross-correlation between the image and the filter, and extracts the results from central part of the whole results with the same size as the image.

3.3.2 Convolution

Cross-correlation can be done by a convolution with the filter rotated 180 degrees as mentioned in Chapter 2. There are 2 main steps doing 3D image filtering using convolution:

1) Firstly, we rotate the Gabor filter by indexing, which is in MATLAB syntax:

```
    h = h (end: -1: 1, end: -1: 1, end: -1: 1).
```

2) Then we use the built-in function, `convn`, to convolve the 3D CT image with the 3D Gabor filter:

```
    result = convn (image, h, ‘same’).
```

For a standard convolution, the size of a complete calculation results is size (results) = size (image) + size (filter) – 1, but the ‘same’ option indicates that the results are extracted from the central part with the same size as the image.

3.3.3 FFT based convolution

According to the Convolution Theorem mentioned in Chapter 2:

convolution (image, filter) = IFFT (FFT(image) .* FFT(filter)),

The convolution can be calculated by taking FFTs, and the main progress in details is described below:

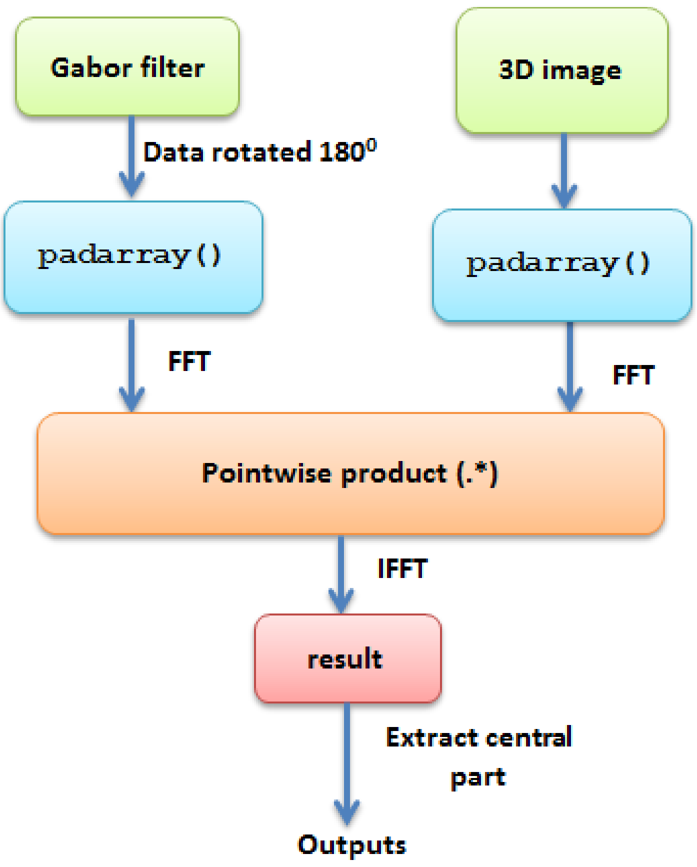


Figure 3.5: The workflow of FFT based convolution

1) Firstly, the data of Gabor filter are rotated by 180 degrees.

2) We pad zeros at the end of the 3D image and the 3D filter using the function `padarray()`, which is:

```
    image = padarray (image, size(filter)-1, 0, 'post'),

    filter = padarray (filter, size(image)-1, 0, 'post').
```

It expands the size to size (image) + size (filter) – 1, so that the circular period of FFT covers all the convolution results.

Alternatively, it is common to pad the arrays to a power of 2 using the function `nextpow2` and `padarray`, in order to achieve higher performance for doing FFTs [22]. Because the MATLAB FFT function is based on the FFTW, and the white paper of CUFFT suggests this [22].

However, the 3D CT image is a large data set, which occupies much memory, and is easy to exceed the limitations of 2.7 GB on Fermi. For example, if we pad the 512 * 512 * 300 CT image with zeros to the size of a power of 2, the expanded size will become 1024 * 1024 * 512 which will take 4GB space for double precision real data, 8GB for complex data, and 16GB for computing point-wise product between two

FFTs. It costs too much memory and thus does not make it particularly suitable for GPUs. Therefore, the size was just expanded to size (image) + size (filter) – 1.

3) 3D FFTs are performed on the expanded image and filter using the function `fftn`.

4) We make a point-wise product between the two previous FFTed data.

5) An inverse 3D FFT is performed to the product from the previous step:

```
    result = ifftn ( fftn (image) .* fftn (filter) ).
```

6) The results are extracted from the central part of the total results by indexing:

```
    head = floor (size (filter) / 2 + 1);

    rear = head + size (image) - 1;

    result = result ( head(1): rear(1), head(2): rear(2),
head(3): rear(3) ).
```

3.3.4 FFT based cross-correlation

The FFT based cross-correlation works in a similar procedure as the FFT based convolution, but there is no need to rotate the Gabor filter. Instead, we do a conjugation on one of the FFTs, according to the theory mentioned in Chapter 2:

correlation (image, filter) = IFFT (FFT(image).*conjugation (FFT(filter))).

The main procedure of FFT based cross-correlation is described below:

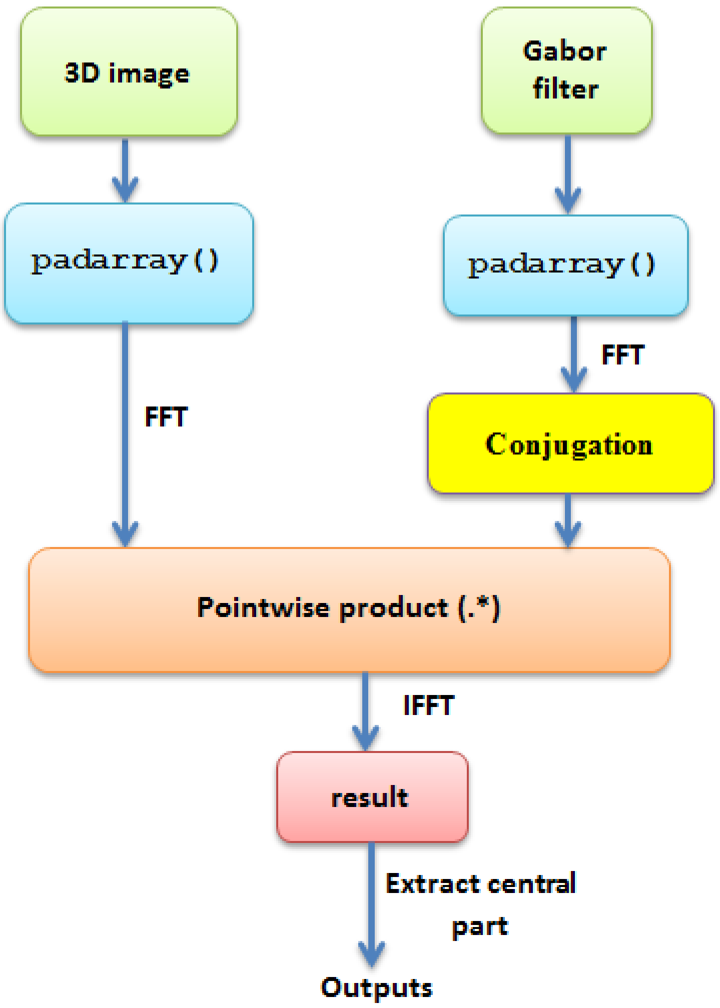


Figure 3.6: The workflow of FFT based cross-correlation

1) The 3D image and filter are padded with zeros to the size of size (image) + size (filter) – 1, the same as for the FFT based convolution.

2) 3D FFTs are performed on the expanded image and filter using the function `fftn`.

3) We conjugate the FFT results of the filter.

4) We make point-wise product between the conjugation in the 3rd step and the FFT of the filter.

5) An inverse 3D FFT is performed to the product produced in the 4th step, and the code is shown below:

```
    result = ifftn (fftn (image) .* conj (fftn (filter)) ).
```

6) The results are extracted from the central part of the total results as the FFT based convolution.

The FFT based cross-correlation has higher performance than the FFT based convolution, because doing conjugation is an atomic operation which is far more efficient than doing rotations on a 3D array.

3.4 MATLAB Parallel Toolbox with Multiple CPU Cores

Although this project focuses on optimisation with GPUs, we still implemented a parallelisation with multiple CPU cores, whose results were used to compare with the GPU computing.

There are two additional components used for running MATLAB code in parallel with multiple CPU cores. One is the MATLAB Parallel Computing Toolbox (PCT), and the other is the MATLAB Distributed Computing Server [9]. These two official add-ons enable MATLAB to do parallel computing, and we can parallelise the 3D texture analysis code on MATLAB. The Parallel Computing Toolbox provides some methods to parallelise MATLAB codes, for example, parallel for loops, SPMD (single program multiple data), message passing functions, distributed arrays, and so on [11]. The Distributed Computing Server enables MATLAB codes to run on a cluster, and provides the ability to use job management tools from the toolbox.

Prior to parallelising the original code, we have to make sure that each iteration process is independent of the others. There are 144 iterations in the application of Gabor filtering, and each iteration process generates a different Gabor kernel. For each iteration process, the original CT image is filtered by the different Gabor kernel, and gives out different results. Therefore, all the iterations are independent and can be parallelised.

Parallelising the 3D texture code using the MATLAB PCT is our first parallelisation approach, and it was finished at a very early stage, so it was parallelised directly without implementing the algorithm improvement described in the previous section.

The MATLAB PCT provides a quite simple method for CPU parallelisation, and we use the construct `parfor` to parallelise the loops in the part of the Gabor filtering [9]. There are three loops nested in the original code for Gabor filtering as shown below:

```
for i = 1:length(F)

   for j = 1:length(psi)

        for k= 1:length(phi)

            [h,g] = gabor(F(i),B(i),psi(j),phi(k),1);

             …

          end

     end

end
```

However, the construct `parfor` cannot be nested [9]. Therefore, we try to combine the three loops into one loop, in order to make sufficient parallelisation. We can extend the arrays of `F`, `B`, `psi`, and `phi`, responding to the relevant loops, and then we can run a combined loop on the extended arrays. After that, we can parallel all the iterations in a `parfor` loop. The construct `parfor` can separate the tasks to different workers on either a shared memory system or a distributed memory system. Moreover, the `parfor` loop distributes all the iterations automatically to the workers. Using the `parfor` loop is very straightforward, we replace the `for` loop with a `parfor` loop. The optimised code is shown below:

```
	F_tmp = repmat(F, 1, length(psi)*length(phi));

	B_tmp = repmat(B, 1, length(psi)*length(phi));

	psi_tmp = repmat(psi, length(F), length(phi));

	phi_tmp = repmat(phi, length(F)*length(psi), 1);

	% Combine the 3 nested loops.

	parfor iter = 1: (length(F)*length(psi)*length(phi))

	[h,g]=gabor(F_tmp(iter),B_tmp(iter),psi_tmp(iter),

	      phi_tmp(iter),1);

	…

	end
```

In order to run a parallelised MATLAB code, the MATLABpool has to be opened and configured [9]. The MATLABpool connects to a pool of workers, and creates a parallelised job on the pool. We should open a MATLABpool before running a parallel code, and close the pool after the code ends; otherwise it keeps occupying the multiple cores. The testing script in MATLAB is written as below:

```
	MATLABpool (12); % Open a pool with 12 workers

	load('../data/test_volumes','test_volume_64_64_37')

	begin=tic;

test_results_64=calc_3d_stats(test_volume_64_64_37,64);

	t_64=toc(begin);

	clearvars test_vol*

	MATLABpool close;
```

The procedure of using MATLAB PCT to parallelise the original code is very straightforward, and the parallelised code can be run on the local machine Fermi0 and the cluster of Indy. Indy will be introduced in the next chapter. The results obtained are presented in Chapter 4.

3.5 MATLAB Parallel Computing Toolbox with GPU

MATLAB provides its PCT for parallel computing as mentioned in the previous section, and it is simple for programmers to use for parallelising and accelerating a serial code [9]. The PCT not only supports parallelisation with multiple CPUs, but also provides supports for GPU acceleration.

It provides a function named `gpuArray` that can transfer CPU data to GPU data from the host to the device, and uses the function *gather* to return the GPU data to CPU data from the device to the host [11].

First, we tried to use the MATLAB built-in functions to accelerate the image filtering with GPUs. MATLAB has some built-in functions that support GPU computing, but the function `imfilter` is not available for 3D data on GPUs even in the latest version of MATLAB (at the time of writing this was R2013a).

Another function `convn`, which is multiple dimensional convolution, is only available on the latest version MATLAB R2013a, and is not available on the version R2012a which was what was deployed on Fermi0 [8].

The PCT also provides a method `arrayfun` that enable programmers to write their own GPU codes [11]. However, the `arrayfun` has strict limitations that can only support basic calculation code, and it cannot even support the `for` loops on GPUs. So it has difficulties to filter 3D images using the `arrayfun`.

Finally, we also made use of FFT based convolution and FFT based cross-correlation for GPU acceleration. The GPU acceleration with the FFT based convolution is described below as an example:

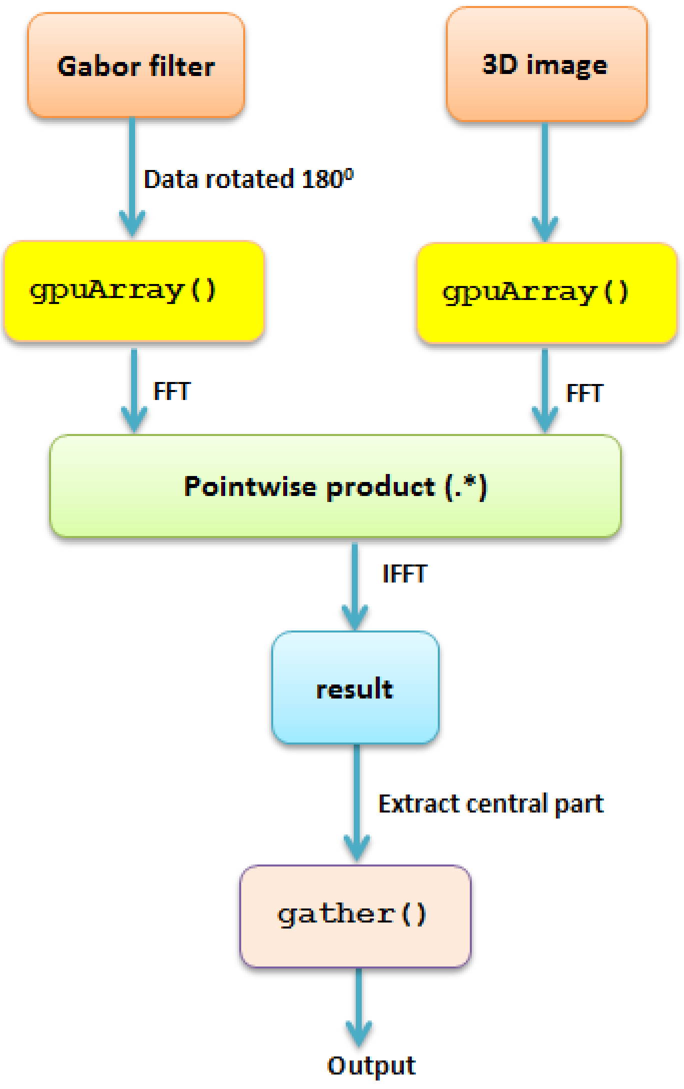


Figure 3.7: The workflow of the FFT based convolution with PCT GPU

1) First, the data of Gabor filter should be rotated 180 degrees.

2) The 3D image and the 3D filter should be transferred from the MATLAB working space to a GPU, using the `gpuArray`:

```
    gpu_img = gpuArray (image);

    gpu_filter = gpuArray (filter);
```

3) The 3D image and the 3D filter are padded with zeros through the use of the function `fftn`. Due to the limitation of the GPU memory on Fermi0 (2.7GB), the size of the FFTs cannot be a power of 2. For example, if we pad the 512 * 512 * 300 CT image with zeros to the size of a power of 2, the expanded size will become 1024 * 1024 * 512. It will take 4GB space for double precision real data, 8GB for complex data, and 16GB for computing point-wise product between two FFTs. It uses too much memory, and exceeds the GPU memory. Therefore, the size is just expanded to size (image) + size (filter) – 1.

4) The 3D FFT based convolution is performed as below:

```
result = ifftn ( fftn (image, size) .* fftn (filter, size) ).
```

5) The results then can be extracted from the central part of the total results as the serial FFT based convolution code.

6) Finally, the GPU results are transferred back to the MATLAB working space, using the function `gather`:

```
     result = gather (result).
```

Although the Parallel Computing Toolbox currently only provides limited supports for GPU computing, its FFT functions for GPUs have already been highly optimised, and performed very well. The FFT functions in MATLAB are based on the FFTW and CUFFT [12], which have the very top performance in the FFT implementation on CPUs and GPUs respectively. For this project, the image filtering mostly relies on the FFTs, so it has already provided great help on the GPU acceleration to the 3D texture analysis code.

The MathWorks Company has realised the power and importance of GPU computing, and their Parallel Computing Toolbox is being improved to support GPUs. There are more and more built-in functions that support GPU arrays being included in recent years [8]. In addition, the MathWorks has bought a GPU add-on Jacket from AccelerEyes, and is integrating the highly optimised GPU add-on Jacket into the Parallel Computing Toolbox. The PCT will have more supports and higher performance on GPU computing in the near future. Results obtained using this optimisation is discussed in Chapter 4.

3.6 Jacket

Jacket is a third party add-on from the AccelerEyes Company [13], and it provides greater supports and higher performance for GPU computing than the current MATLAB Parallel Computing Toolbox. Where the PCT has 171 built-in functions supporting GPU data, Jacket has provided 589 GPU functions [15]. Jacket not only provides a wider range of GPU functions, but also provides better acceleration with GPUs. Jacket has been tested that its GPU functions have been optimised and have higher performance than the PCT, according to a number of benchmark reports [16]. In addition, Jacket is much easier for GPU programming than CUDA or OpenCL.

In addition, Jacket was purchased from AcclerEyes by the MathWorks in December 2012 [13], and is being integrated into MATLAB Parallel Computing Toolbox. Jacket cannot be purchased any more but existing installed versions are being supported for a year. For this project, Jacket was still available on Fermi0 at version 1.8.2.

In order to run a Jacket code with MATLAB, the path of the Jacket engine has to be added to MATLAB using the command "addpath <jacket_root>/engine". On Fermi0, the "<jacket_root>" is "/usr/local/jacket/". We need to add the path of Jacket before running a Jacket code, and then MATLAB can work with Jacket and find its libraries.

The path of Jacket will be removed after the code ends. We write a testing script to run the Jacket code in MATLAB, which is shown below:

```
addpath /usr/local/jacket/engine

addpath /usr/local/jacket/sdk

load('../data/test_volumes','test_volume_64_64_37')

begin=tic;

test_results_64 =
      calc_3d_stats(test_volume_64_64_37,64);

t_64=toc(begin);

clearvars test_vol*

rmpath /usr/local/jacket/engine

rmpath /usr/local/jacket/sdk
```

Jacket provides two functions named `gdouble` and `gsingle` that transfer double and single precision respectively CPU data to the GPU from host to device, and in reverse, the function `double` and `single` return the double and single precision GPU data to CPU from device to host [13].

Taking a similar approach to using the PCT with GPUs, we first tried to use the Jacket built-in functions to accelerate the image filtering with GPUs. Although Jacket has as many as 589 built-in functions that support GPU computing, the function `imfilter` is not available for 3D data on GPUs, so we came across the same situation as with the PCT [13].

Another function `convn`, which is a multiple dimensional convolution function, is available in Jacket [14], but we were more interest in FFT based convolution which can run much faster to produce a result. Therefore, the Jacket acceleration code with GPUs is based on the algorithm of FFT based convolution. The procedure of the FFT based convolution with GPUs is similar to the PCT approach, and is described below as an example:

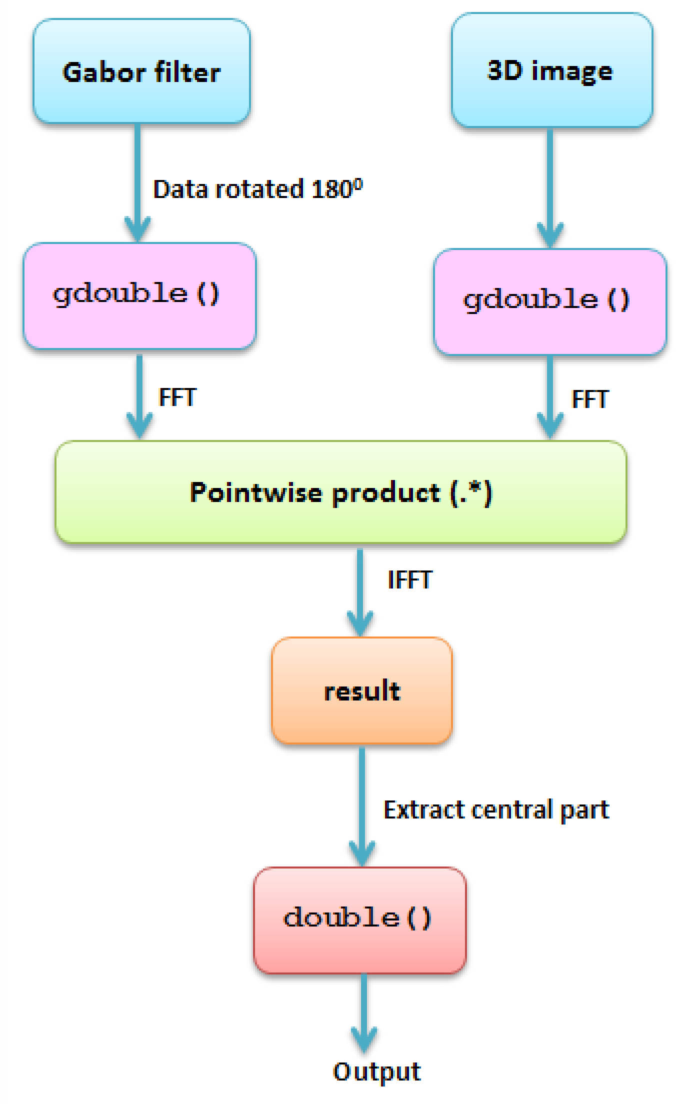


Figure 3.8: The workflow of the FFT based convolution using Jacket

1) First, the data of Gabor filter should be rotated 180 degrees.

2) The data of the 3D image and the 3D filter should be transferred from host to device, and the data type should be transformed from `double` to `gdouble` using the `gdouble` function:

```
    gpu_img = gdouble (image);

    gpu_filter = gdouble (filter);
```

3) Just like with the PCT strategy, the 3D image and the 3D filter are padded with zeros through the use of the `fftn` function. Due to the limitation of the GPU memory on Fermi0 (about 2.5GB), the size of the FFTs cannot be a power of 2. It uses too much memory, and exceeds the limitation of the GPU memory. Therefore, the size is just expanded to size (image) + size (filter) – 1.

4) The 3D FFT based convolution is performed as below:

```
result = ifftn ( fftn (image, size) .* fftn (filter, size) ).
```

5) The results then can be extracted from the central part of the total results as the PCT approach.

6) Finally, the GPU results are transferred back to the MATLAB working space, using the function `double`:

```
     result = double (result).
```

In addition, Jacket has its own parallel for loops on GPUs named `gfor`, which offers significant advantages over the PCT mentioned in the previous section. The PCT has no GPU for loops. The Jacket `gfor` loops can automatically apply data on the GPU, and executes calculation in the GPU. However, the `gfor` loop does not help improve performance of the code in the Gabor filtering, when we use it for the 144 independent iterations of the Gabor filtering. The `gfor` loops can be applied in other parts (e.g. Haralick features) for further optimisation in the future.

The Jacket FFT is also based on the CUFFT (CUDA FFT library), which have the very top performance in the FFT implementation on NVIDIA GPUs. Thus, the Jacket FFT functions are also highly optimised for GPU computing as the PCT, and accelerate the 3D texture analysis code greatly.

Since the image filtering mostly relies on the FFTs in this project, many other GPU functions in Jacket have not been used. So the many advantages of Jacket are not so obvious in this project, but Jacket should still help improve the performance with GPUs in other parts of the code – a task for future work.

3.7 CUDA

The Compute Unified Device Architecture (CUDA) is a mature GPU programming model developed by the NVIDIA Company [17]. For NVIDIA GPU cards, CUDA gives the highest performance and flexibility for GPU programming, but it is much more difficult to program and requires much more effort than using the PCT or Jacket. We have to consider the data transfer between the host and device, 3D data memory access, number of threads, and many other relevant programming details.

In order to develop CUDA codes, we need a CUDA enabled GPU card, the CUDA toolkit, and its relevant driver. Fermi0 already had the NVIDIA CUDA Toolkit installed, and also had the CUDA-capable driver for its Tesla C2050 GPU card.

CUDA has its own FFT library named CUFFT [22]. The CUFFT library is in based on the FFTW, which is one of the most efficient CPU implementation of FFT [22]. The FFT uses a divide-and-conquer algorithm for efficiently calculating discrete Fourier transform.

The CUFFT provides a simple API for parallel FFT implementation on a GPU card. It uses a handle named a “plan” to configure internal building blocks and GPU hardware resources before computing the FFTs, so that it can reduce the execution time as much as possible [22]. The pre-configuration has advantages in that, once the plan has been created, it can be retained there and can be executed multiple times without re-configuration. After setting the plan, the execution function can then takes place according to the plan. This CUFFT mechanism works well, because it configures internal threads and GPU resources for different kinds of FFTs, and then can execute FFTs several times according to the plan. It makes good uses of the GPU resources [22].

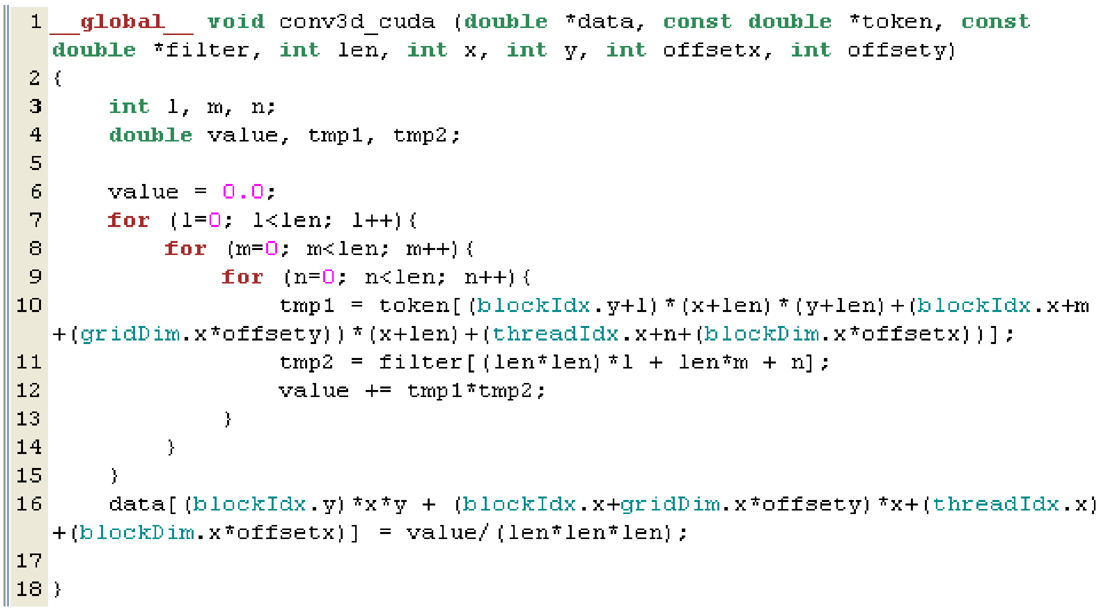
CUDA code can be called from within MATLAB. MATLAB supports hybrid programming with other languages, such as C and CUDA [21], in order to get acceleration for part of the MATLAB code. We can select the most computational demanding part of the original code which is the image filtering, and implement this using CUDA.

There are two methods calling CUDA codes from within MATLAB. One uses the PTX files, the other uses the MEX files [20]. Both of them are described respectively in the following section, and are compared to each other.

3.7.1 MATLAB Calls CUDA through PTX Files

Although MATLAB has some supports for GPU computing, many functions still could do with more optimisation, such as the `imfilter`. MATLAB can call a CUDA kernel through Parallel Thread Execution (PTX) files [19]. This section describes the workflow of making a kernel from CU and PTX files.

1) A CUDA kernel code is written in a CU file. Because the CUDA kernel code cannot interact with host-side libraries, for example CUFFT, it cannot thus use the CUDA FFT library [19]. Therefore, we implement a basic 3D convolution instead, shown in Figure 3.9, which is not as good as an FFT based convolution.



```
__global__ void conv3d_cuda (double *data, const double *token, const
double *filter, int len, int x, int y, int offsetx, int offsety)
{
    int l, m, n;
    double value, tmp1, tmp2;

    value = 0.0;
    for (l=0; l<len; l++){
        for (m=0; m<len; m++){
            for (n=0; n<len; n++){
                tmp1 = token[(blockIdx.y+l)*(x+len)*(y+len)+(blockIdx.x+m
+(gridDim.x*offsety))*(x+len)+(threadIdx.x+n+(blockDim.x*offsetx))];
                tmp2 = filter[(len*len)*l + len*m + n];
                value += tmp1*tmp2;
            }
        }
    }
    data[(blockIdx.y)*x*y + (blockIdx.x+gridDim.x*offsety)*x+(threadIdx.x)
+(blockDim.x*offsetx)] = value/(len*len*len);

}
```

Figure 3.9: The CUDA kernel code of a basic 3D convolution

The directive "__global__" indicates that it is an entry point of the CUDA kernel. The code implements a basic convolution according to the definition of convolution, so its performance is not as good as the FFT based convolution.

2) We can use the NVIDIA compiler (`nvcc`) to compile the CUDA kernel code (CU file) at the shell command line, and generate a PTX file, using the following command:

```
     nvcc –ptx testgabor.cu
```

3) Then we create a kernel in MATLAB using the PCT.

```
k=parallel.gpu.CUDAKernel('testgabor.ptx','testgabor.cu');
```

4) At last, we can set the number of blocks and threads, and run the kernel in MATLAB using the function `feval`:

```
     N=64;

     k.ThreadBlockSize = [N N N];

     result = feval (k, image, filter, …);
```

This example is an implementation of a basic convolution, which is not efficient enough. It is relatively easy to use the PTX files to call a CUDA kernel, but it has the limitation that it cannot use host-side library. Therefore, in the next section, we implement an FFT based convolution with CUFFT using the MEX files to call CUDA code, which has much better acceleration.

3.7.2 MATLAB Calls CUDA through MEX Files

The method of calling CUDA using MEX files is much more complicated than the method of using PTX files, but MEX files has more flexibility that it can interact with the host-side libraries [20]. In other words, we have to use MEX files to call CUDA code if we want to make uses of the CUFFT (CUDA FFT library).

The MEX file is a MATLAB API used for interfacing CUDA code, or other languages, to MATLAB functions [21]. In this way, MATLAB is able to be extended through MEX files taking advantage of the computational power offered by CUDA and GPUs.

This work shows the feasibility and benefits of using MEX files to call a CUDA code, which implements an FFT based convolution. The workflow describes the procedure of accelerating MATLAB with CUDA using MEX files, which is shown in Figure 3.10:

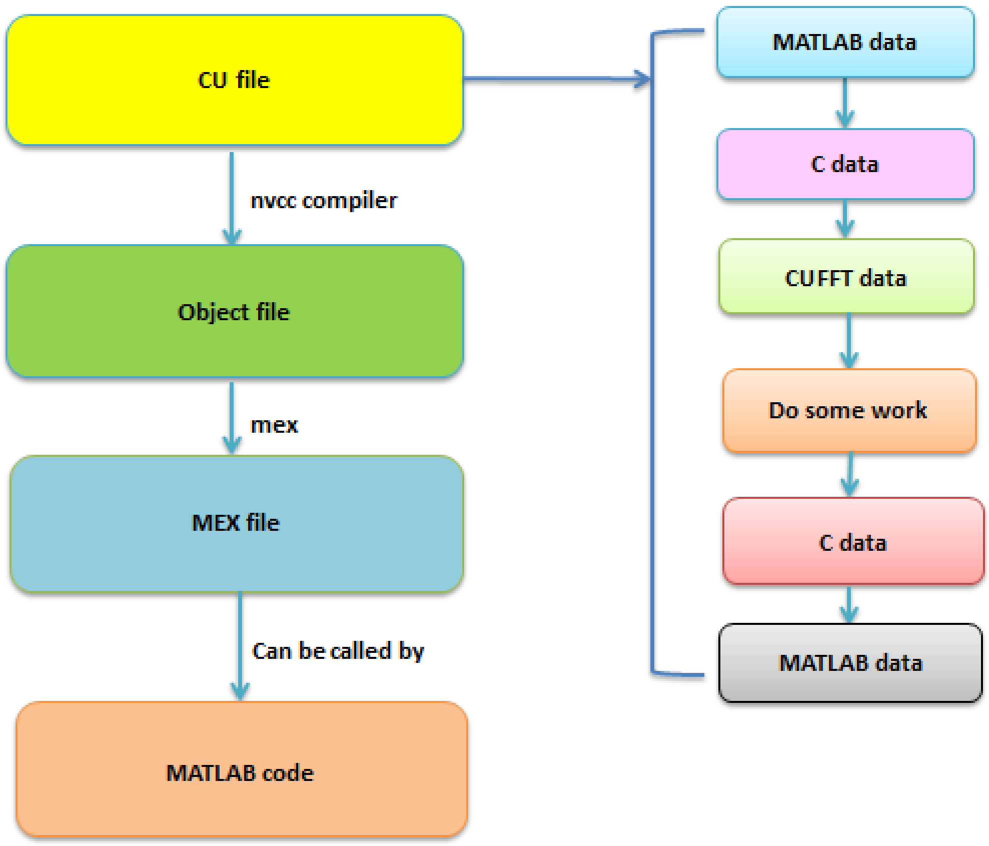


Figure 3.10: The workflow describes the procedure of accelerating MATLAB with CUDA using MEX files

1) There are some header files which have to be included in the source CUDA code:

```
     #include <cuda_runtime.h>

     #include <cufft.h>

     #include "mex.h"

     #include "matrix.h"
```

The "cufft.h" is used for the CUFFT, and the "cuda_runtime.h" is a common header file for CUDA codes. It is necessary to include the "mex.h" which is the interface between the MEX file and CUDA code. Lastly, the "matrix.h" is for the MATLAB data types, which will be converted into C data types.

2) The gateway function in a MEX file is `mexFunction`, which is the entry point for MATLAB to access the CUDA code. The `mexFunction` is shown below:

```
   mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
mxArray *prhs[ ])

     {
        ...

        }
```

The mexFunction is equivalent to the *main* function in C code. The parameter nlhs is the number of output MATLAB arrays (left hand side). The parameter plhs is the array of pointers to the expected outputs. The parameter nrhs is the number of input arrays (right hand side). The parameter prhs is the array of pointers to inputs.

3) Variables should be declared in the CU source file as shown in Figure 3.11.

```
mwSize n, nDim;
const mwSize *dimSize;
int nx, ny, nz, i, j, k, id;
double *A, *C_real, *C_img;
cufftDoubleComplex *C;

cufftHandle plan;
double *gpuA;
cufftDoubleComplex *gpuC;
```

Figure 3.11: Variables declared in the CUDA code

The CUFFT has complex value data type cufftDoubleComplex, and we create a plan using the cufftHandle.

4) The input MATLAB data are converted into C data, as shown below:

```
n = mxGetNumberOfElements(prhs[0]);
nDim = mxGetNumberOfDimensions(prhs[0]);
dimSize = mxGetDimensions(prhs[0]);
A = mxGetPr(prhs[0]);

plhs[0] = mxCreateNumericArray(nDim, dimSize, mxDOUBLE_CLASS, mxCOMPLEX);
C_real = mxGetPr(plhs[0]);
C_img  = mxGetPi(plhs[0]);
```

Figure 3.12: Getting input MATLAB data to C data

The n is the number of elements of the input 3D arrays, and “nDim” gets the number of dimensions, which is 3 here. The “dimSize” is the size of each dimension, and the A is the pointer of the input 3D array. The “plhs[0]” is the pointer to the output 3D array. The “C_real” is the pointer of the real part of the output complex array, while the “C_img” is the point of the imaginary part of the outputs.

The complex value array is stored in two separate arrays in C.

5) The 3D data layout should be re-ordered, because the mechanisms of memory access are different between the MATLAB and C [23]. The MATLAB arrays are column-major, while C arrays are row-major. The ordering of 3D data arrays should be adjusted as shown in Figure 3.13. The complex data are stored in C_real and C_img separately, and the id is a counter.

```
id = 0;
for (k = 0; k < nz; k++) {
    for (j = 0; j < ny; j++) {
        for (i = 0; i < nx; i++) {
            C[i*ny*nz + j*nz + k].x = A[id];
            C[i].y = 0.0;
            id++;
        }
    }
}
```

Figure 3.13: Converting the ordering of 3D data arrays

6) We allocate memory on the GPU using the CUDA function `cudaMalloc`, as described below:

```
cudaMalloc((void**)&gpuA,sizeof(cufftDoubleComplex)*nx*ny*
nz);

cudaMalloc((void**)&gpuC,sizeof(cufftDoubleComplex)*nx*ny*
nz);
```

7) The C data is transferred from the host to the device using the CUDA function `cudaMemcpy` as shown below:

```
    cudaMemcpy(gpuC,C,sizeof(cufftDoubleComplex)*nx*ny*nz
, cudaMemcpyHostToDevice);
```

8) Then we can create a 3D FFT plan, which configures the internal blocks and threads of the GPU resources. The plan is created as described below:

```
    cufftPlan3d(&plan, nx, ny, nz, CUFFT_Z2Z);
```

9) After the plan has been created, we can execute the 3D FFT as shown below:

```
    cufftExecZ2Z(plan, gpuC, gpuC, CUFFT_FORWARD);
```

10) We also do other calculations on the GPU, for example, point-wise product.

11) The GPU data is transferred back from the device to the host using `cudaMemcpy`.

12) The results of 3D data layout should be re-ordered back. The ordering of 3D data arrays should be adjusted as shown in Figure 3.14:

```
id = 0;
for (i = 0; i < nx; i++) {
    for (j = 0; j < ny; j++) {
        for (k = 0; k < nz; k++) {
            C_real[i+j*nx+k*ny*nx] = C[id].x;
            C_img[i+j*nx+k*ny*nx]  = C[id].y;
            id++;
        }
    }
}
```

Figure 3.14: Re-order the 3D data layout

13) Finally, the memory should be cleaned up, and the plan should be cleaned up:

```
cufftDestroy(plan);

cudaFree(gpuA);

cudaFree(gpuC);
```

The workflow shown above is the description of writing the CUDA code (CU file) with CUFFT. We have to compile the CU file into a MEX file at the next stage. The compilation is separated into 2 steps. The first step is to use the NVIDIA compiler nvcc to compile the CU file into an Object file, and the second step uses the MATLAB mex script to compile the Object file into a MEX file (mexa64 for Linux 64).

In the MATLAB command window, we type the two lines of commands shown below:

```
!nvcc -c cudaFFT3d.cu -Xcompiler -fPIC -I /usr/local/MATLAB/extern/include

mex cudaFFT3d.o -L /usr/local/cuda/lib64 -lcudart -lcufft
```

The first line of command is a shell escape with a "!" at the start, while the second command line has no "!". At last, the MEX file cudaFFT3d.mexa64 is generated, and can be called from MATLAB.

From the description of using MEX files with CUDA, we can find that it is complicated because of the data conversion. The MATLAB data is first converted to C data, and the indexing needs to be re-ordered for 3D arrays. Then the C data is transferred to the GPU, and the CUFFT has a specific complex data type. The CUFFT only supports interleaved format for complex data, while C stores all the real data followed by the imaginary data. The data conversion is complicated and one should be careful.

3.7.3 Comparison between MEX files and PTX files

MEX files provide a powerful API for MATLAB to call other language code, such as CUDA, C, or FORTRAN. Using MEX files can invoke CUDA codes on the GPU and to handle data transfer between the host and device. On the other hand, the use of PTX files provides an easier way to call a CUDA kernel, although it has some limitations.

There are two main advantages of using MEX files:

- MEX files are more flexibility. They can interact with host-side libraries, like CUFFT, and also with the CUDA runtime library.
- MEX files can allocate memory, and configure the internal blocks and threads.

3.8 Comparison of the Different Approaches

After the different kinds of optimisation to the original 3D texture analysis code were implemented, a comparison is given in this section.

Improving the algorithm for the image filtering requires effort in understanding the underlying theory (and reading papers to gain this understanding). We have to learn some theories of image analysis and to understand the original texture analysis code. Furthermore, it takes quite a lot of time to do some research by reading papers. On the other hand, algorithm improvement brings huge acceleration.

The optimisation using PCT with CPUs needs to combine the nested loops into one loop, because the `parfor` cannot be nested. It takes the least effort in this project.

The optimisation using PCT with GPUs is quite straightforward, because MATLAB has provided a well optimised 3D FFT function for GPUs. The optimisation using Jacket has the similar situation to the PCT with GPUs. The implementation of these two approaches is easier than CUDA, and can get very high performance as the CUDA implementation.

CUDA is the most challenge approach to optimise the 3D texture analysis code. It costs the most time and efforts among all the approaches. Since we want to use CUFFT, we have to use the MEX files to call CUDA. The method of using MEX files requires many data conversions, which are quite complicated. The data types are different among MATLAB, C, and CUFFT (complex data type). The 3D arrays layout is also different between MATLAB (column-major) and C (row-major). Furthermore, we have to consider the memory access and data transfer between the host and device. The approach of using CUDA needs much more programming efforts than the other approaches.

This chapter presents the profiling, design and implementation of the optimisation to the original code, including algorithm improvement, multiple CPU-cores parallelisation, and GPU acceleration. The next chapter will illustrates and analyses the results and speedup, and compares the performance among different approaches, different testing sizes and different machines Fermi0 of Hydra and Comp002 of Indy.

Chapter 4

Results and Analysis

This chapter reports and analyses the results obtained for the different optimisation strategies applied to the original 3D texture analysis code.

4.1 Machines and Platforms

The different versions of optimised codes were tested on the backend node Fermi0 of Hydra and the backend node Comp002 of Indy.

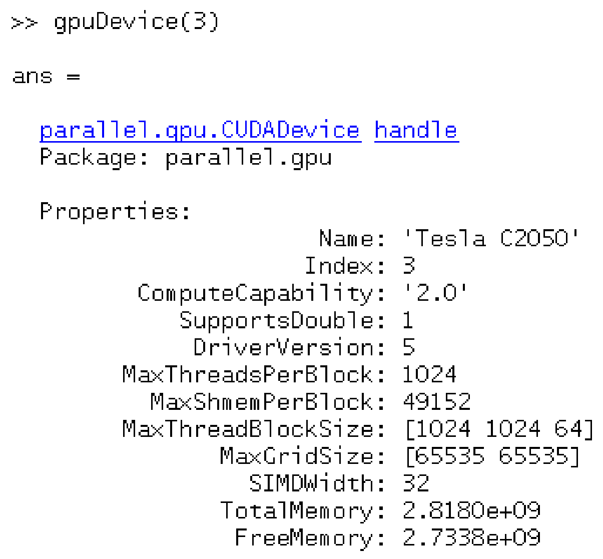
Fermi0 is a workstation which with 4 Intel Xeon X5650 2.67GHz CPUs, each CPU has 6 cores. Fermi0 also hosts 4 NVIDIA Tesla C2050 GPU cards. Its operating system is Linux x64, and has MATLAB R2012a installed on it.

The other machine Comp002 on Indy has 4 AMD Opteron 6276 2.24GHz CPUs, each CPU has 16 cores. Moreover, Comp002 has a GPU of NVIDIA Tesla K20. Its operating system is also Linux x64, and MATLAB R2013a is installed I it.

There are 4 NVIDIA Tesla GPU cards on Fermi0. When a MATLAB GPU code executes it chooses the first one by default. If the first GPU card on Fermi0 is busy, it will exit, and we can then select another device, using the PCT function *gpuDevice*. For example, we can select the 3rd one by indexing:

gpuDevice (3).

If the device is available, MATLAB will show the device information as shown below.



```
>> gpuDevice(3)

ans =

  parallel.gpu.CUDADevice handle
  Package: parallel.gpu

  Properties:
                      Name: 'Tesla C2050'
                     Index: 3
         ComputeCapability: '2.0'
            SupportsDouble: 1
             DriverVersion: 5
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [65535 65535]
                 SIMDWidth: 32
               TotalMemory: 2.8180e+09
                FreeMemory: 2.7338e+09
```

Figure 4.1: GPU device information

The performance of the original 3D MATLAB code is shown in Table 4.1. The Gabor filtering takes up most of the total time, and the optimisation effort here focuses on this. The other parts of the code only spend a little proportion of the total time, thus they have been left as is in the serial code.

Image Size	*Total time(sec)*	*Gabor (sec)*	*Other routines (sec)*
64 * 64 * 37	776	771	5
128 * 128 * 75	6473	6450	23
256 * 256 * 150	53043	52846	197
512 * 512 * 300	425011	423470	1541

Table 4.1: The timing results of the Gabor filtering and other parts

4.2 Checking Correctness

It is crucial to check the correctness when optimising a code. We compare the results between the optimised codes and the original code. The relative error is defined as follows:

$$relative\_error = \frac{Rn - Ro}{Ro}$$

where Rn is the new results, and Ro is the original results.

We start by verifying the texture features calculated by the multi-core code version (PCT with CPUs). All the output results are exactly the same to the original results. This indicates that the optimised code using PCT with CPUs works correctly.

For the codes using improved algorithms, we compare their outputs with the original results. Their relative errors are less than 10^{-15}, which are probably due to the double precision rounding off. For example, for the FFT based convolution of the 64 * 64 * 37 test case CT image the results of all the Gabor features are of order 10^{36}, and the absolute errors (Rn-Ro) are about 10^{21}. So the relative errors are $10^{21} / 10^{36} = 10^{-15}$, which are probably double precision rounding-off. Figure 4.2 shows the relative errors between the results of the FFT code and the original code, along the Gabor features.

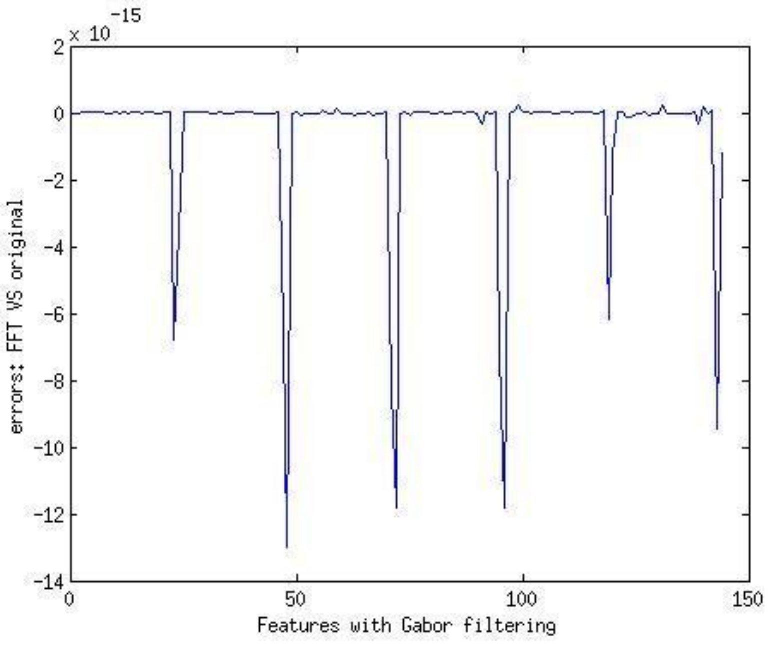


Figure 4.2: Relative errors between the results of the FFT code and the original one

From Figure 4.2, it is clear that the errors along the Gabor features are periodic. In other words, relatively big errors happen every 24 Gabor features. The reason why this is the case is still not clear.

For the GPU codes, the results are compared with the relevant CPU results. Their relative errors are also less than 10^{-15}, which again are probably due to the double precision rounding off. Figure 4.3 shows the relative errors of the results between the GPU FFT code and CPU FFT version, along the Gabor features or the GPU FFT based convolution with the 64 * 64 * 37 test case CT image.

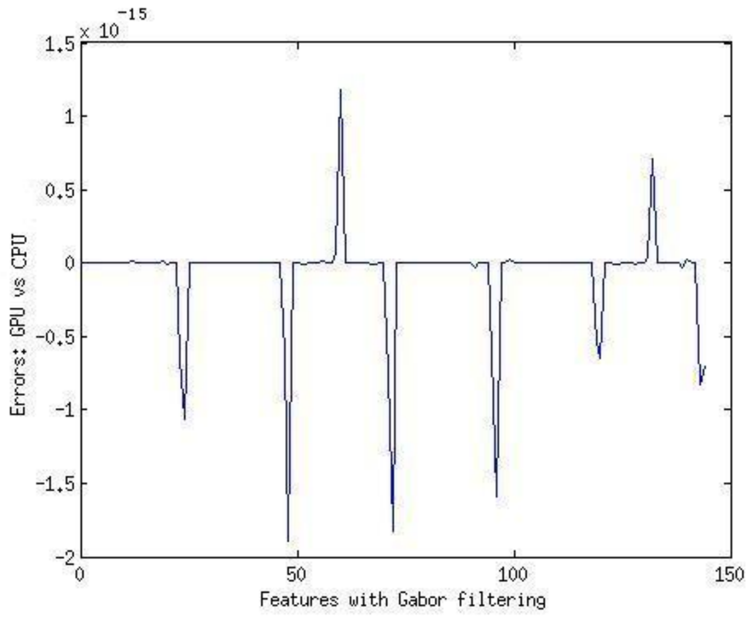


Figure 4.3: Relative errors of the results between the GPU FFT code and the CPU FFT version

Figure 4.3 is similar to Figure 4.2. The errors along the Gabor features are also periodic. Relatively big errors happen every 24 Gabor features. Furthermore, the errors of the original code results between Fermi0 and Comp002 are shown in Figure 4.4.

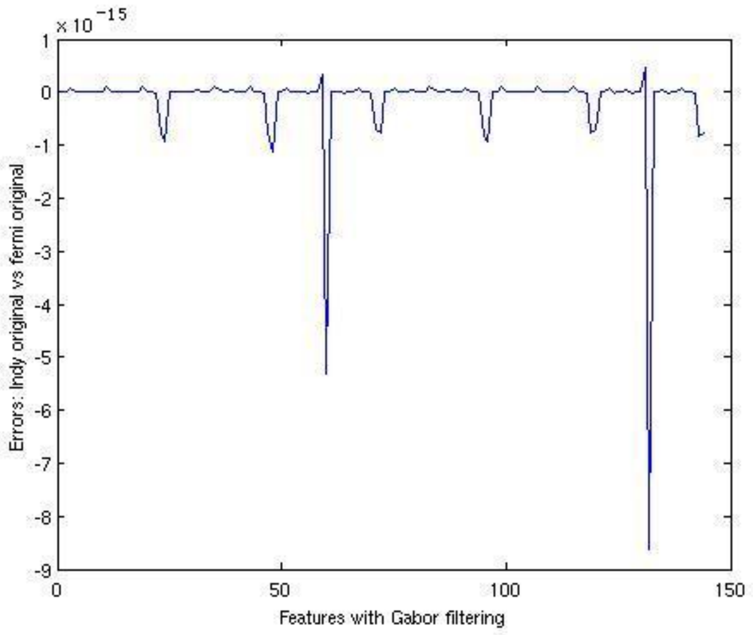


Figure 4.4: Relative errors of the original results between Fermi0 and Comp002

4.3 Performance of the Different Algorithms

We have optimised the image filtering by using different algorithms, and now we compare their performance for the 4 different test cases. The serial image filtering codes which are compared are:

- The original code using `imfilter`
- Image filtering using `convn`
- Image filtering using an FFT based convolution
- Image filtering using an FFT based correlation

The 4 test cases are 3D CT images with different sizes: 64 * 64 * 37, 128 * 128 * 75, 256 * 256 * 150, and 512 * 512 * 300.

The speedup is defined as

$$\text{speedup} = T_{old} / T_{new},$$

where T_{old} is the execution time of the original code, and the T_{new} is the execution time of the optimised code.

4.3.1 Acceleration of the Image Filtering

The speedup of different algorithms for the part of image filtering is illustrated in Figure 4.5.

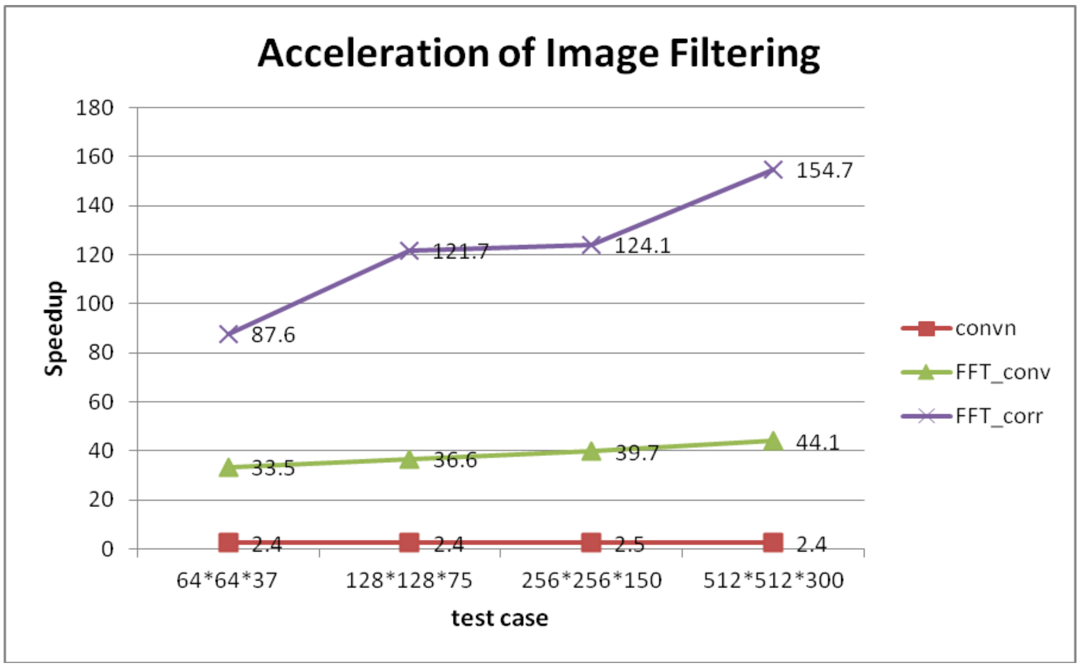


Figure 4.5: Speedup of the image filtering part of the original code by using different algorithms

For the image filtering part, using the MATLAB convolution function `convn` has resulted in a speedup of 2.4x for all the four test cases. This indicates that the MATLAB built-in function `convn` has more optimisations than the function `imfilter`. From the source code of `imfilter` and `convn`, `imfilter` works in a direct way as the basic definition, while the `convn` has the ability to separate a 3D kernel into 3 1D kernels. The `convn` judges whether a multiple-dimensional kernel is separable at the start, if it is, it will separate the 3D kernel and perform a 1D convolution 3 times. This will reduce the calculation complexity, and reduce the execution time. In the image filtering code, there are also Gaussian kernels, which are separable, so using the `convn` can get some speedup, although it spends extra time on separating kernels.

The FFT based convolution produces a much larger speedup than using the `convn` routine. It has got more than 30 times faster than the original code, and even gets an increasing speedup as the test cases increase in size. The FFT based convolution can largely reduce the calculation complexities as mentioned in Chapter 2. Moreover, the MATLAB FFT functions are based on FFTW, which has the top performance for FFT implementation.

The FFT based correlation is the fastest among the image filtering algorithms. It gets best speedups of all: 87 times speedup for the 64 * 64 * 37 CT image, and 154 times speedup for the 512 * 512 * 300 test case. The larger a data set is, the more acceleration it achieves. It is faster than the FFT based convolution, because it performs a conjugation rather than a rotation as required for FFT based convolutions. Conjugation is an atomic operation for computers, while a rotation consumes memory and spends time on memory access. The FFT based correlation has achieved a significant acceleration to the original code.

4.3.2 Acceleration of the Complete System

We would also like to know what the benefit to the whole code execution is by improving the Gabor filter algorithm.

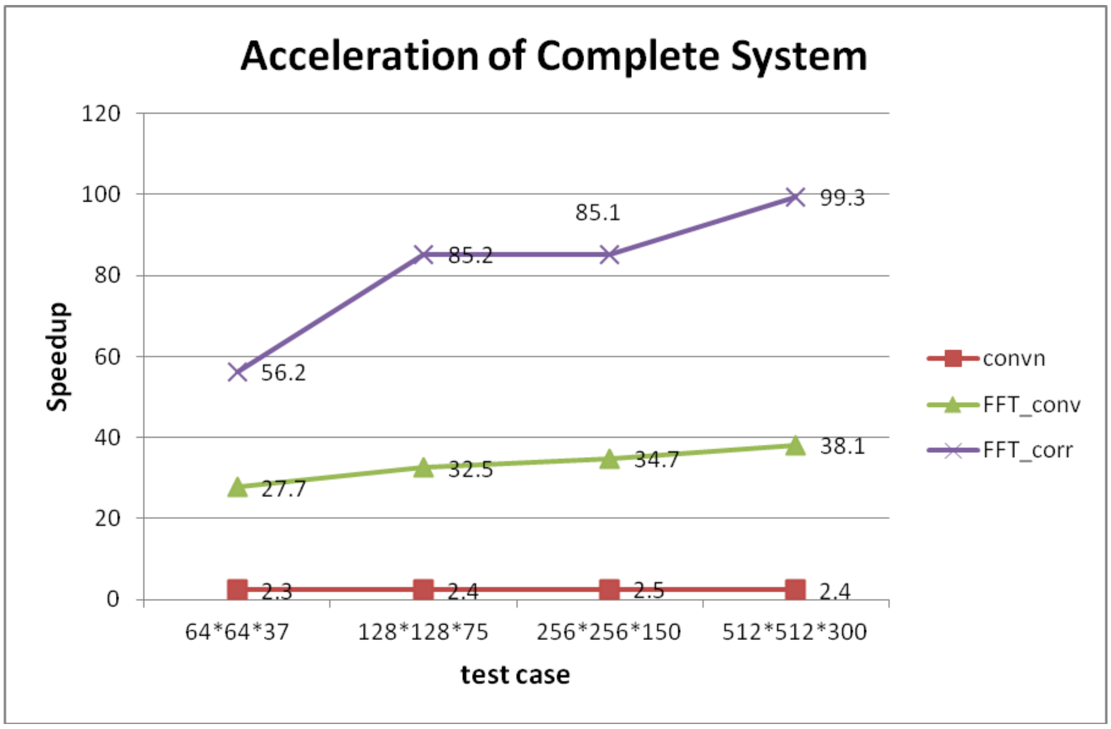


Figure 4.6: Speedup of the complete system

The MATLAB `convn` function gives a steady speedup of 2.4, while the FFT based algorithms provide increasing speedup with the increasing size of test cases. That is because the FFT based algorithms reduce the calculation complexity. Assuming the image size is m * m * m, and the filter size is n * n * n, the FFT based algorithms have a calculation complexity of O $((m+n)^3 \log (m+n))$, while the `imfilter` method is O (m^3n^3). When m and n increase, the advantage of FFT based algorithms is obvious. In this project, the FFT based convolution has 27.7 times speedup for the 64 * 64 * 37 test case, and a 38.1 times speedup for the 512 * 512 * 300 test case. The FFT based cross-correlation produces an even higher acceleration, a 56 times speedup for the 64 * 64 * 37 test case, and 99 times speedup for the largest test case (512 * 512 * 300).

Figure 4.6 shows the speedup for the whole system, and the performance comparison for using the different algorithms.

4.4 Speedup with Multiple Cores (CPU)

The benchmarking of the multi-cores code was done on the Comp002 node of Indy, as the Indy cluster allows up to 64 cores to work in parallel. However, the Indy cluster is so busy that we could only get up to 24 cores (also named labs in MATLAB), and could open a `matlabpool` of 64 or even 36 cores.

The benchmarking used 3 test cases: 64 * 64 * 37, 128 * 128 * 75, and 256 * 256 * 150. The largest case costs too much time to run. We calculated the speedup and efficiency show in the figures below.

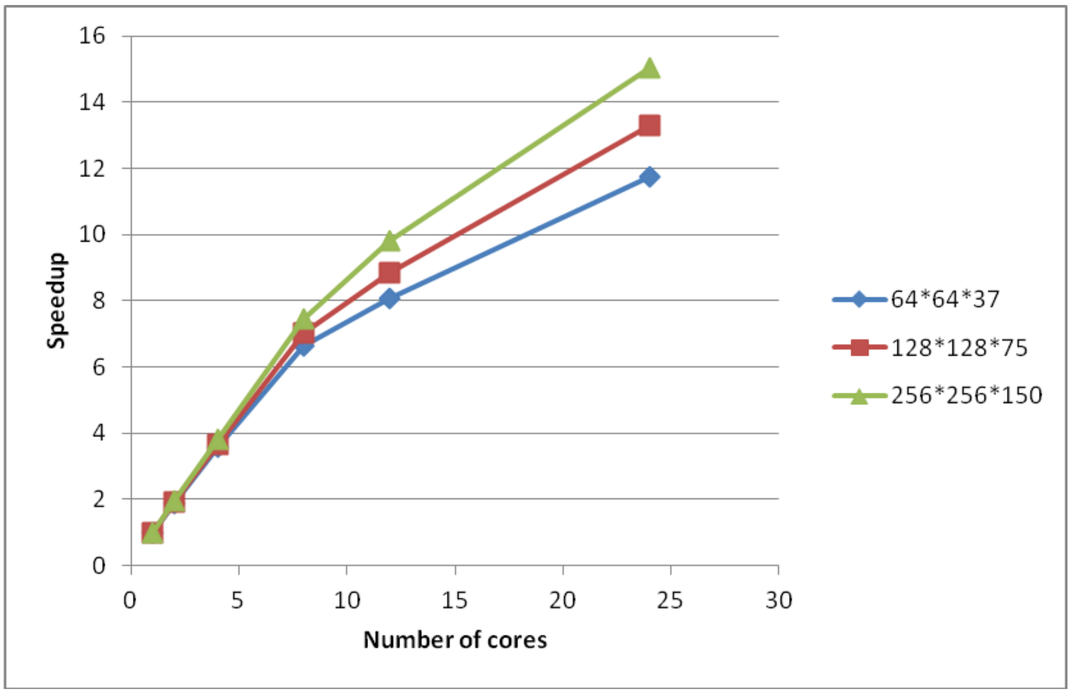


Figure 4.7: Speedup with different number of CPU cores

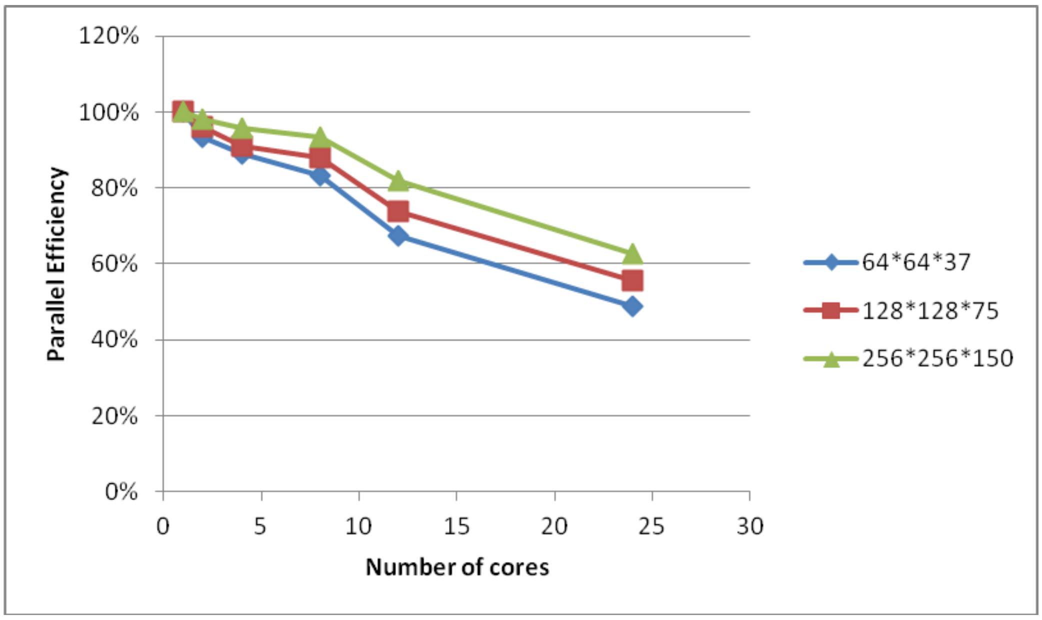


Figure 4.8: Parallel efficiency with different number of CPU cores

The speedup for different number of CPU cores is shown in Figure 4.7. The time running on 1 core for each case was 776 sec, 6473 sec, and 53043 sec respectively. The code runs faster as the number of cores increases. In other words, parallelisation with multiple cores brings improvements to the performance of the code. In addition, when benchmarking as the problem size is increased, the speedup still improves. That means the code can still run much faster in parallel as the input test cases become larger.

The parallel efficiency with different number of CPU cores is shown in Figure 4.8. The parallel efficiency decreases as the number of cores increases. This is because the code has serial part, and there are parallelisation overheads, such as jobs scheduling. In addition, we get higher parallel efficiency when the testing size becomes larger.

According to the Figure 4.7 and Figure 4.8, the speedup is better and the efficiency is higher, when the input image has a larger size. That is because as the input image size becomes larger, the serial part of the code and the parallelisation overhead are proportionally relatively small. When the testing size is small, the serial part and parallelisation overhead will affect the total execution time. Therefore, parallelisation with multiple CPU cores is more appropriate in the test cases with larger size.

4.5 Speedup with GPUs

This section tests the performance of the three GPU codes on Fermi0 and Comp002. The three GPU codes were optimised using PCT, Jacket and CUDA. All the three GPU codes are based on the FFT convolution algorithm, as the GPU codes were implemented before I became aware of the FFT cross-correlation method which is more efficient than the FFT convolution used here. The FFT cross-correlation algorithm port to the GPU codes has been logged as future work.

4.5.1 The Speed of Data Transfer

In order to have a better understanding of the GPU performance, it is necessary to measure the speed of data transfer between the MATLAB working space and the GPU. Table 4.2 shows the results:

Machine	*MATLAB to GPU (GB/s)*	*GPU to MATLAB (GB/s)*
Fermi0	2.92	1.78
Comp002	2.17	0.41

Table 4.2: The speed of data transfer between the MATLAB working space and the GPU

The results shown in Table 4.2 were timed using the MATLAB timer (`tic` and `toc`), since the GPU codes are implemented on the platform of MATLAB. The speed of data transfer between MATLAB and GPU contains two parts: one is data transfer between host and device, and the other is data conversion between the MATLAB working space and the GPU.

If the data size is small, the overhead of data transfer between the host and device is relatively large. If the data size is large, it will cost much time on the data conversion between the MATLAB working space and the GPU.

The speed of data transfer is higher on Fermi0 than Comp002. The possible reason for this is that the PCI-E interface on Comp002 has lower bandwidth than Fermi0.

The speed of the data transfer from the GPU to MATLAB is slower than that from MATLAB to the GPU. That is because the MATLAB memory manager spends more time allocating dynamic memory in its working space, while the GPU has more efficient memory allocation.

4.5.2 Measuring Acceleration with GPUs

Due to the limitation of the GPU memory, the GPU codes using the FFT algorithms can only run the 3 test cases: 64 * 64 * 37, 128 * 128 * 75, and 256 * 256 * 150. For the largest test case, if we pad the 512 * 512 * 300 CT image with zeros to the size of a power of 2, the expanded size will become 1024 * 1024 * 512 which will take 4GB space for double precision real data, 8GB for complex data, and 16GB for computing point-wise product between two FFTs. The testing is taken place on both Fermi0 and Comp002. Comp002 does not have Jacket installed.

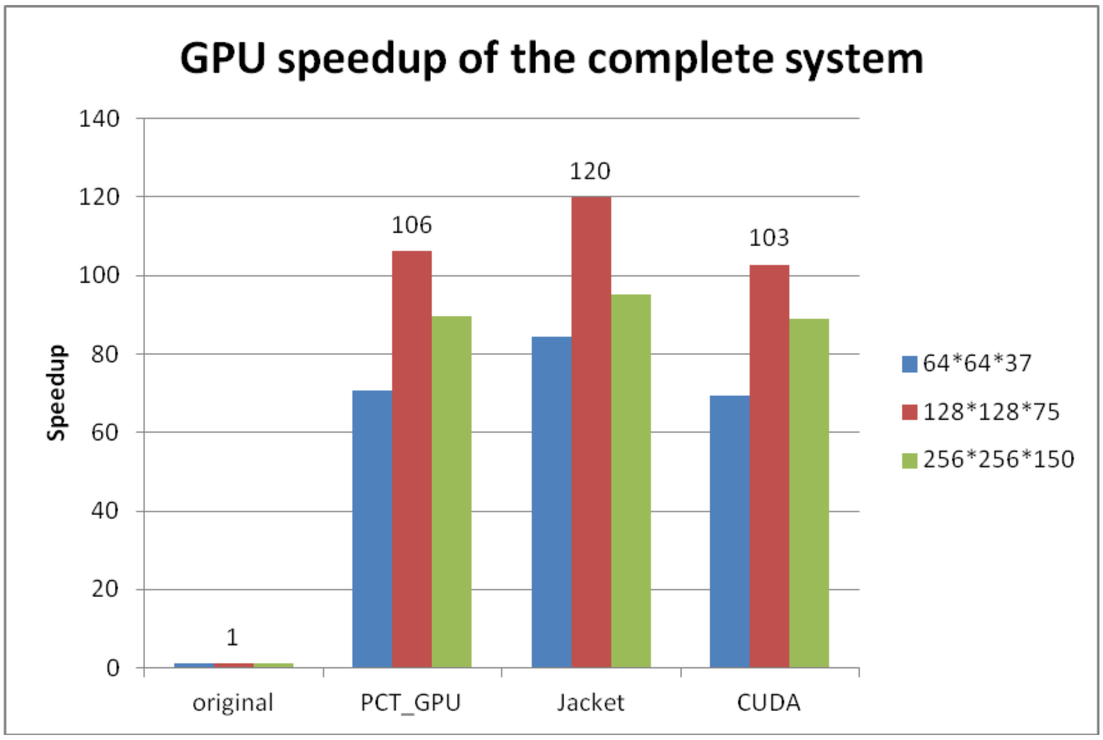


Figure 4.9: The GPU speedup of the complete system on Fermi0

Figure 4.9 illustrates the speedup of the complete system with GPUs on Fermi0. The time of the original code running on 1 core for each case was 776 sec, 6473 sec, and 53043 sec respectively. Compared with the original code, the GPU code using PCT has a speedup from 70 to 106. The Jacket code can achieve a speedup from 80 to 120, and the CUDA code can get a speedup from 70 to 103.

All the three GPU codes have similar performance with each other. That is because the FFT convolution mostly relies on the FFT, and all the GPU codes are based on the CUDA FFT library (CUFFT), so they have similar performance.

In addition, the speedup in the 256 * 256 * 150 test case is lower than the 128 * 128 * 75 test case, because the GPU codes have too much data transfer, which is a huge overhead and should be optimised in further work.

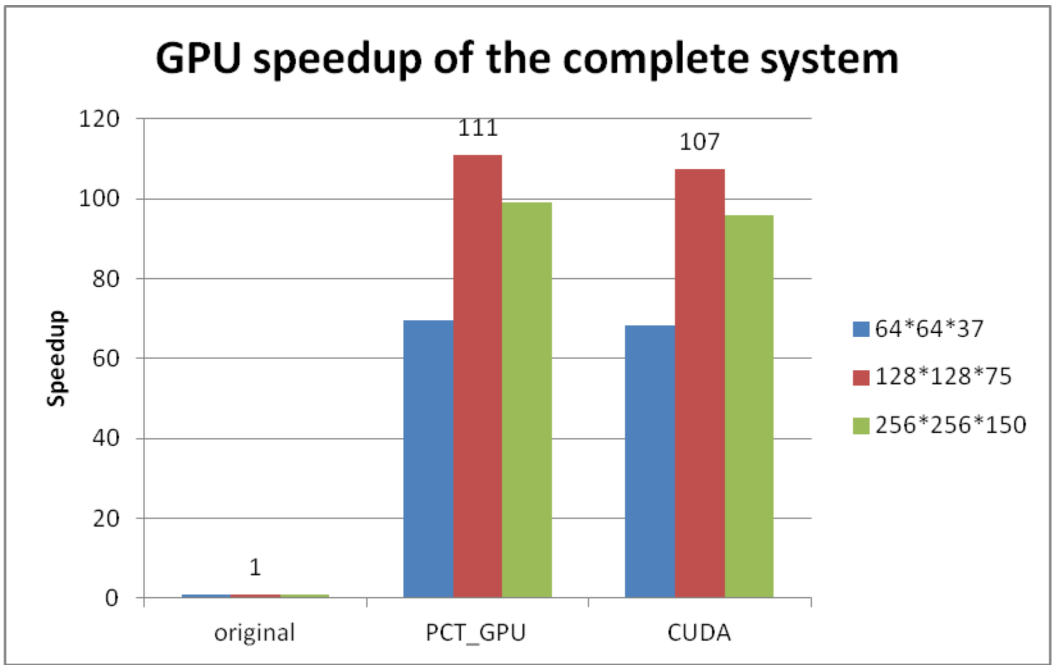


Figure 4.10: The GPU speedup of the complete system on Comp002

Figure 4.10 illustrates the speedup of the complete system with GPUs on Comp002. The time of the original code running on 1 core for each case was 776 sec, 6473 sec, and 53043 sec respectively. Compared with the original code, the GPU code using PCT has a speedup from 70 to 111 and the CUDA code can get a speedup from 70 to 107.

The two GPU codes have similar performance, because they are both based on the CUDA FFT library (CUFFT). In addition, the GPU performance is slightly better on Comp002 (Tesla K20) than on Fermi0 (Tesla C2050). Because the GPU codes have too much data transfer, which is a large overhead affecting the performance, so it should be optimised in future works.

4.6 Performance of all Approaches

Finally, we compare the performance of all the different approaches. Figure 4.11 and Figure 4.12 show all the speedup measurements obtained on Fermi0 and Comp002 respectively.

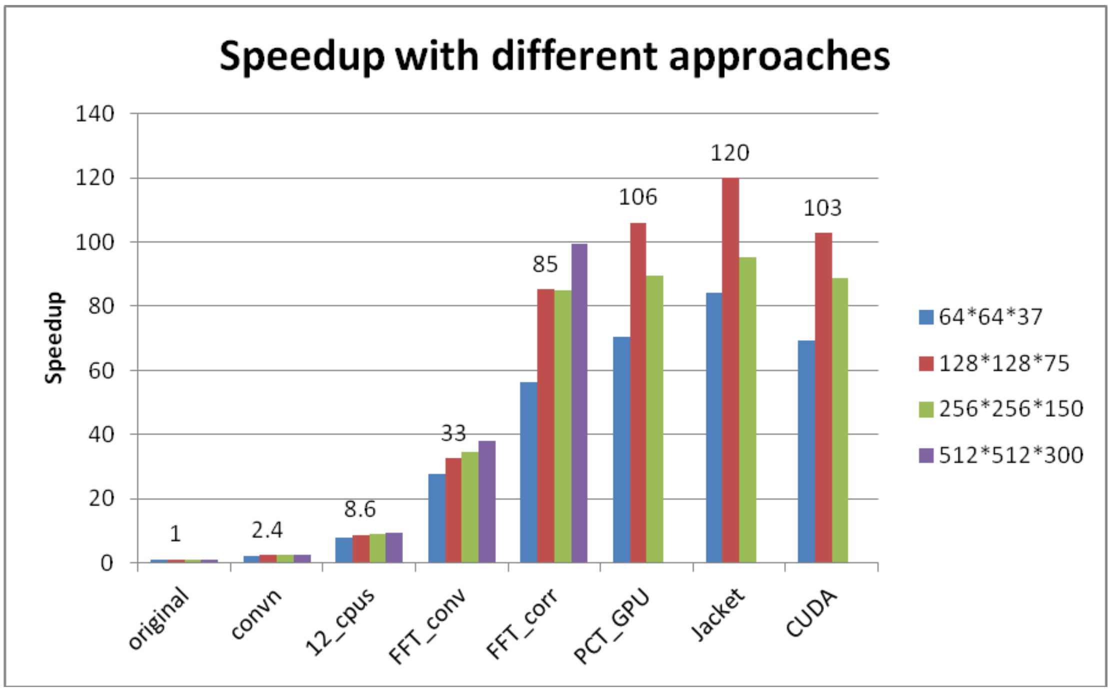


Figure 4.11: The speedup of all approaches on Fermi0

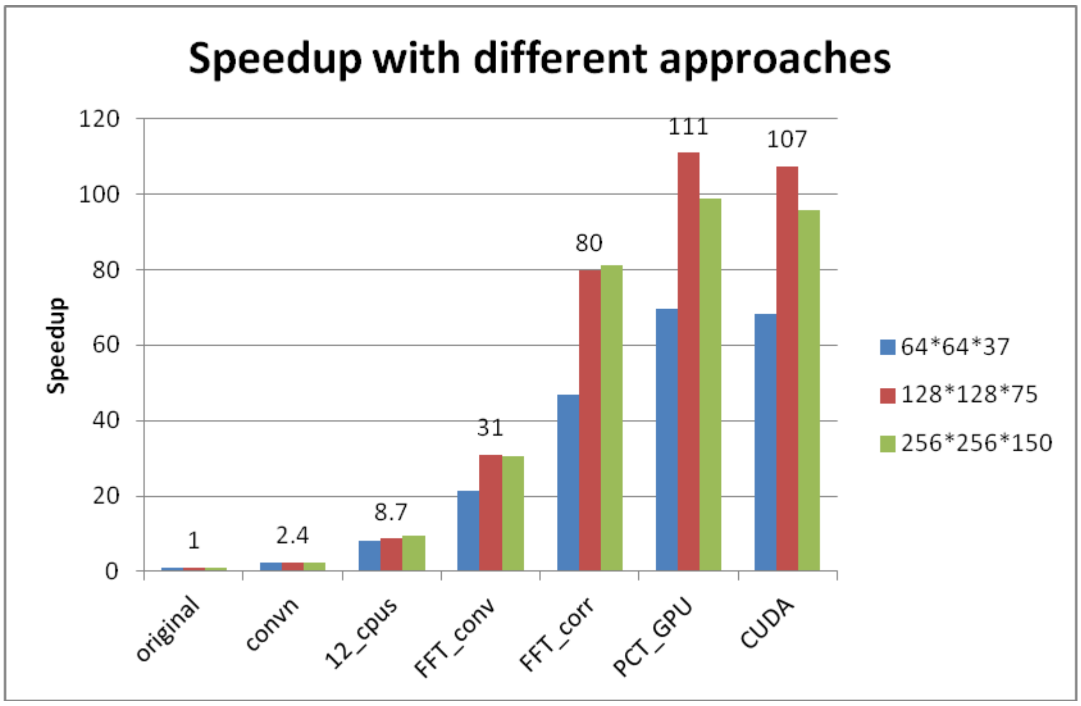


Figure 4.12: The speedup of all approaches on Comp002

Both of the two machines produce similar speedup results. The algorithm improvements alone achieve a high speedup which is further enhanced by the parallelism obtained by the hardware used. Particularly, the FFT cross-correlation CPU serial code achieves a speedup from 57 to 80. However, this was developed at a late stage of this work, and there was not enough time to port it to GPU codes. The GPU codes with FFT convolution algorithm can achieve more than 100 times speedup.

This chapter illustrates and analyses the results and speedup, and compares the performance among different approaches, different testing sizes and different machines. In the next chapter, we will discuss a conclusion and further work.

Chapter 5

Conclusions

This dissertation investigated various approaches to optimising a three dimensional texture analysis code, provided by the Cancer Research Centre at the Western General Hospital in Edinburgh. The challenge of this project has been to process large data sets of three dimensional CT images. The 3D texture analysis code is much more computationally demanding than the 2D code used in another MSc last year [6]. In addition, the image filtering with Gabor filter which was not in the 2D code, is the most computational part in the 3D code taking more than 99% of the total execution time. Therefore, this project focused on optimising the Gabor filtering, and accelerated its performance using a number of different approaches.

The implementation and testing were done on the backend node Fermi0 of Hydra and the backend node Comp002 of Indy. Fermi0 has 4 GPUs of NVIDIA Tesla C2050 cards, and Comp002 has a GPU of NVIDIA Tesla K20. The development environment was MATLAB. There were 4 3D test case CT images with sizes: 64 * 64 * 37, 128 * 128 * 75, 256 * 256 * 150 and 512 * 512 * 300.

This project aimed to accelerate the original MATLAB code using different kinds of approaches, and focused on how to process large data set in a faster way. We improved the image filtering algorithm, and applied both multi-cores and GPU parallelisation techniques. The original MATLAB code was optimised largely by the algorithm improvements and GPU techniques, and has achieved a large speedup of 120 times.

For algorithm improvement, we have tried: convolutions (a MATLAB built-in function), FFT based convolutions, and FFT based cross-correlations. The MATLAB convolution had a 2.3 time speedup. The algorithm of FFT based convolution run 27 times faster than the CPU serial code. Furthermore, we also improved the transform from convolution to correlation, using conjugation approaches instead of rotating three-dimensional arrays, which benefited the memory usage and reached 57 times faster than the original code.

After that, we used the MATLAB Parallel Computing Toolbox (PCT) to parallelise the original code. The multi-cores code got a reasonable speedup, and its performance results were used to analyse the speedup from using different numbers of cores. Based on the original algorithm, the parallelised code can get 8 times speedup when it is run on 12 cores.

We then ported the FFT based convolution code to the GPU based codes. The first GPU technique we used was the MATLAB Parallel Computing Toolbox (PCT). The PCT provides some support for GPU computation. It provides the FFT built-in functions that support GPU computation, which have performed very well in this case and are based on CUFFT [22]. The GPU based code using PCT run more than 100 times faster than the original code.

The second GPU technique we applied was Jacket, a product of AccelerEyes. It is a GPU add-on for MATLAB. It has done deep optimisations for GPU calculation, and it provides a much easier way for GPU programming compared to using CUDA in MATLAB. Jacket achieved the highest performance (a speedup of 120) in this project, as its FFT performs well. The CUDA code called by MEX files has too much data (complex data types) conversion between MATLAB, C and CUDA. In going from each of these to another, a data conversion was involved.

The third GPU technique used CUDA to try to accelerate the original MATLAB code. CUDA has a professionally optimised FFT library for GPU computation named CUFFT, which gives great performance and convenience. The FFT functions in PCT and Jacket are based on CUFFT as well [12, 16]. However, the CUDA code called by MEX files has too much data (complex data type) conversion taking place between MATLAB, C and CUDA, and thus does not have a high efficiency.

There are a number of potential performance improvements for the 3D texture analysis code that there was insufficient time to do during this project and could be done in future work. The data transfer is not well optimised in the current GPU codes. In addition, the FFT based cross-correlation algorithm, which has much higher performance than the FFT convolution, has not been ported to the GPU codes yet. The GPU codes can get much higher speedup with the FFT cross-correlation algorithm. Finally, we could also try to develop a multiple GPUs code with MATLAB PCT, using `parfor` loops. If this work were to be progressed it would be good to try some of these things.

References

[1] D. Montgomery, K. Cheng, Y. Feng, D. McLaren, S. Erridge, S. McLauglin, S. Campbell, and W. Nailon, *Predicting the Occurrence of Radiation Induced Pneumonitis by Texture Analysis of CT Images from Lung Cancer Patients*, Western General Hospital, University of Edinburgh, Heriot Watt University, 2013.

[2] Y. Xu, M. Sonka, G. McLennan, G. Junfeng, and E. Hoffman, *MDCT-based 3-D texture classification of emphysema and early smoking related lung pathologies*, IEEE Trans. Med. Imag., vol. 25, no. 4, pp. 464-475, 2006.

[3] V. Kovalev, F. Kruggel, H. Gertz, and D. von Cramon, *Three-dimensional texture analysis of mri brain datasets*, IEEE Trans. Med. Imag., vol. 20, no. 5, pp. 424-433, 2004.

[4] A. Kurani, D. Xu, J. Furst, and D. Raicu, *Co-occurrence matrices for volumetric data*, in 7th IASTED Int. Conf on Computer Graphics and Imaging, 2004.

[5] Luis Cebamanos, *A GPU-based platform for cancer-treatment planning*, EPCC MSc dissertation, 2011.

[6] Dante Gama Dessavre, *Using High Performance Computing to Improve Image Guided Cancer Treatment*, EPCC MSc dissertation, 2012.

[7] Ahmed Adnan Aqrawi, *Three Dimensional Convolution of Large Data Sets on Modern GPUs*, Norwegian University of Science and Technology, 2009.

[8] MATLAB official webpage, http://www.mathworks.co.uk/ , last accessed date 17/08/2013.

[9] MATLAB Parallel Computing Toolbox, http://www.mathworks.co.uk/products/parallel-computing/, last accessed date 17/08/2013

[10] MATLAB imfilter, http://www.mathworks.co.uk/help/images/ref/imfilter.html, last accessed date 17/08/2013.

[11] MATLAB gpuArray, http://www.mathworks.co.uk/help/distcomp/using-gpuarray.html#bsloua3-1, last accessed date 17/08/2013.

[12] MATLAB FFT, http://www.mathworks.co.uk/help/images/fourier-transform.html, last accessed date 17/08/2013.

[13] Jeremy Fix, *Efficient convolution using the Fast Fourier Transform, Application in C++*, 2011.

[14] AccelerEyes Jacket, http://wiki.accelereyes.com/wiki/index.php/Documentation , last accessed date 17/08/2013.

[15] Jacket learning, http://www.omatrix.com/jacket.html, last accessed date 17/08/2013

[16] Jacket fast computation of cross-correlation, http://wiki.accelereyes.com/wiki/index.php/Fast_Computation_of_Cross_Correlation, last accessed date 17/08/2013.

[17] CUDA documentation, http://docs.nvidia.com/cuda/index.html, last accessed date 17/08.2013.

[18] NVIDIA CUDA, http://www.nvidia.com/object/cuda_home_new.html , last accessed date 17/08/2013.

[19] CUDA PTX files for MATLAB, http://home.agh.edu.pl/~szostek/files/MATLAB_GPU_OLD.pdf, last accessed date 17/08/2013.

[20] CUDA MEX files for MATLAB, http://www.mathworks.co.uk/help/distcomp/create-and-run-mex-files-containing-cuda-code.html#btrgjg_, last accessed date 17/08/2013.

[21] White Paper, *Accelerating MATLAB with CUDA Using MEX Files*, 2007.

[22] NVIDIA, CUFFT Library User Guide, 2012.

[23] Shreyas Vijay Parnerkar, *Memory and Run-Time Efficient Image Texture Classification using NVIDIA GPU as a co-processor*, University of Wisconsin-Madison, 2012.