Medical Image Processing on Intel Parallel Frameworks

Jonathan Low

August 23, 2013

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2013

Abstract

Increases in processor performance are now expected to mainly come from an increasing number of processing cores after clockspeeds reached a limit in 2005. Accompanying this are demands placed on software to become highly parallel and able to scale well to higher thread counts. Investments in time and effort on parallelising software have resulted in applications that exhibit excellent performance and scalability on multisocket CPU nodes. The Intel Xeon Phi product family aims to offer higher performance to those existing highly parallel applications by increasing the thread count well into the hundreds whilst allowing to keep the same programming model. This project follows the proposition of developing a parallel application that scales well on multicore CPU nodes with a view to achieving higher performance on an Intel Xeon Phi co-processor.

This project aims to parallelise a medical imaging application from the Western General Hospital in Edinburgh used to help in the prediction of radiation-induced fibrosis for lung cancer patients after radiotherapy. We use the Intel Parallel Studio XE 2013 software development kit, utilising OpenMP for multithreading and the Intel Math Kernel Library to accelerate application performance on shared memory nodes. The project is then extended to examine the application performance on many more threads on an Intel Xeon Phi.

By a change in the image filtering algorithm utilising Fast Fourier Transforms in addition to parallelisation, we obtain runtimes that are upto 168 times faster over the original code. This was achieved on dual socket Intel Xeon nodes with 48 and 64GB of memory.

Running the same computational code on an Intel Xeon Phi resulted in a levelling of performance after more than sixty threads for the largest image we could run on the co-processor. The 8GB memory limitation prevented the running of a number of threads required for high performance and the runtimes obtained on the Xeon Phi are higher than that achieved on the Intel Xeon compute-nodes.

Contents

List of Tables

List of Figures

Acknowledgements

I would to give my thanks to Mario Antonioletti and Luis Cebamanos for their supervision and guidance for this project. Their help along the way is much appreciated. Also a thank you to Bill Nailon and his students I have met both here at EPCC and at the Western General Hospital.

Chapter 1

Introduction

From a 2011 PRACE survey [1], the two most popular methods to accelerate and parallelise applications is by message-passing programming using MPI and by programming with OpenMP. For distributed memory systems, using MPI is the language of choice whereas for shared memory systems, both MPI and OpenMP can be used, though the latter is the better choice. Both serial and parallel programs using OpenMP can be easily developed and maintained at the same time.

These systems consist of multicore CPU processors such as Intel Xeon, AMD Opteron or IBM PowerPC. The first two are found in today's consumer PC market in desktops and laptops. From 2002 onwards, more and more systems in the Top500[1] consists of these commodity CPUs by Intel and AMD, overtaking the number of systems using proprietary processors [2]. Extracting performance out of these multicore CPUs in parallel has been the key objective for parallel programming languages and APIs such as MPI and OpenMP. One way to achieve this is to multi-thread applications using OpenMP directives that enable execution on multiple cores in parallel using multiple threads.

Overtime, technological and architectural improvements by Intel and AMD have translated to better performance of these CPUs. These can be taken advantage of by existing multi-threaded applications in terms of higher core clockspeeds and increasing core count. The latter now contributes a larger share of increased performance due to reaching thermal limits, causing levelling of processor clockspeeds. The core count on a shared memory node can be increased through use of multiple CPU sockets with multicore CPUs. For example, a node of 64 cores can be constructed using four AMD Interlagos [3], each consisting of 16 cores[2]. For multi-threaded applications that scale well with all 64 cores, usually one predicts that even better performance can be attained if many more of these cores are available.

Intel's Xeon Phi Product Family[3] aims to allow parallel applications, such as those that

[1]http://www.top500/org

[2]In this case, there are 8 modules in total, each consisting of 2 independent cores.

[3]http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

are multi-threaded using OpenMP, to achieve higher performance by simply increasing the number of available cores that run the same x86 instruction set architecture as x86 CPUs such as the Intel Xeons. An Intel Xeon Phi chip can consist of 60 cores[4] for example, each running four virtual threads giving a total thread count of 240. So any parallel application that scales well on shared memory systems with multi-core CPUs should be capable of attaining higher performance with upto 240 available threads.

Achieving parallel performance by using an ever increasing number of x86 cores is in contrast to achieving parallel performance using specialised processors such as the IBM Cell processor [4] or General Purpose Graphics Processing Units (GPGPUs) by Nvidia and AMD[5]. The objective to achieving performance on these processors is to offload many computational routines in parallel onto the floating-point units such as the vector processing units on the Cell or the many simple numerical processing cores in an Nvidia GPGPU. These specialised processors have helped systems to achieve the top spot in the Top500 list. For example the Roadrunner[6], consisting of Cells paired with AMD Opterons, occupied the top spot in the June 2008 list and Titan[7], a Cray system consisting of AMD Opterons paired with Nvidia K20 GPGPUs, reached #1 on November 2012.

Maximising parallel performance from specialised processors such as GPGPUs requires applications to be written using native SDKs. In Nvidia's case, one has to use the proprietary SDK, CUDA, and to follow the data stream processing model suitable for the GPGPU architecture. There is the additional burden of need to maintain two versions of an application, one suitable for CPUs using OpenMP for example and another using CUDA. Languages and APIs such as OpenACC [5] and OpenCL [6] aim to help parallelise applications for CPUs and GPGPUs using compiler directives, however performance is often less than that achievable using native SDKs. Intel's proposition with the Xeon Phi allows new and existing parallel applications using MPI and/or OpenMP that are already well tested and optimised for multicore CPUs to attain further performance by increasing the core count whilst using the same source code.

The project demonstrates the process of attempting to parallelise and accelerate an application using OpenMP to achieve high performance on single compute-nodes consisting of multicore Intel Xeon CPUs. Afterwards, we attempt to attain higher performance by running the same application on an Intel Xeon Phi. We use the Intel Parallel Studio SDK product that consists of compiler, libraries, profilers and debuggers.

The application we accelerate and parallelise is from the Edinburgh Cancer Research Center, located within the Western General Hospital in Edinburgh and is originally written using the MATLAB numerical software package. It is used to help predict whether lung cancer patients will suffer post-radiotherapy treatment side effects, namely radiation-induced pneumonitis after radiotherapy treatment [7]. The application involves the

[4]The number can change between different product variations.

[5]The ATI Radeon product line is now known as AMD Radeon after the purchase of ATI Inc. by AMD in 2006.

[6]http://www.lanl.gov/roadrunner, last accessed on 21-08-2013.

[7]http://www.olcf.ornl.gov/titan, last accessed on 21-08-2013.

processing of image scans of the patient's lungs using a variety of mathematical and statistical techniques. This is a very good application candidate that can make use of parallelisation onto many cores due to the large runtime of the program, taking even days to complete a single processing run for a 3D image of 512x512x300 pixels [8]. The project objectives set out to achieve a runtime far lower than the original program through parallelisation and any algorithmic techniques available.

The dissertation report is divided up as follows:

- Chapter 2 provides the background into the medical imaging application and its aims in helping cancer patients. Emphasis on the aspect of the application that is the most expensive computational part of the application is presented.
- Chapter 3 talks briefly about the Intel Parallel Studio 2013 product suite and the pertinent parts of the SDK that were utilised in this project. Specifications of the hardware used for this project are described here.
- Chapter 4 details the mathematics of the most expensive computational algorithm of the application, namely the correlation and convolution processes. We also describe achieving the same code functionality using the maths libraries provided by Intel in the SDK and present details of linking the developed codes to the MATLAB programming environment. Parallelisation strategies are also presented.
- Chapter 5 examines the performance and scalability of our own developed routines for correlation and convolution using the Intel maths libraries. Benchmarks on single compute-nodes consisting of Intel Xeon processors are presented. This determines the way forward when we integrate the routines with the original application in the MATLAB programming environment.
- Chapter 6 presents the results of improved runtimes obtained over the original application after integration of our parallel routines for the most expensive computational aspect. An analysis of the differences in the answers between our solution and that of the original application is included. Finally, the performance attained on the Intel Xeon Phi and a working framework involving GNU Octave offloading work onto the Xeon Phi is shown.
- Finally, chapter 7 provides conclusions, a summary of the project and suggestions for future work.

Chapter 2

Image Processing & Filtering of Medical CT Scans

The Edinburgh Cancer Research UK Center[1] is a strategic research centre into prevention, diagnosis and treatment of cancer, located at the Western General Hospital (WGH) in North-West Edinburgh and is a collective research partnership between the University of Edinburgh, Cancer Research UK and NHS Lothián. It draws their research from basic clinical trial programs and other theoretical endeavours.

Dr. Bill Nailon and his research team have an application available which utilises a medical image processing code, written in the MATLAB programming language. This code involves the application of statistical algorithms and measurements on medical CT (computed tomography) scans from lung cancer patients. Because the runtime is of the order of hours or even days, this code could benefit from the exploitation of larger and faster computational resources.

This chapter provides the background to the application that we intend to help accelerate through parallelisation. This includes an overview of the application of Gabor filtering in medical applications which is the routine responsible for these long runtimes.

2.1 Radiation-Induced Fibrosis of Lung Cancer Patients

Lung cancer is the second most common cancer in the UK, with no clear single cause for its occurrence although tobacco smoking is often the cited reason for amongst 9 in 10 cases [9]. For lung cancer patients, one of the common approaches to treating lung cancer tumours in patients is by a program of radiotherapy. Unfortunately, such treatment can have side effects, one of which is lung fibrosis induced by the involved radiation.

[1]http://www.ecrc.ed.ac.uk

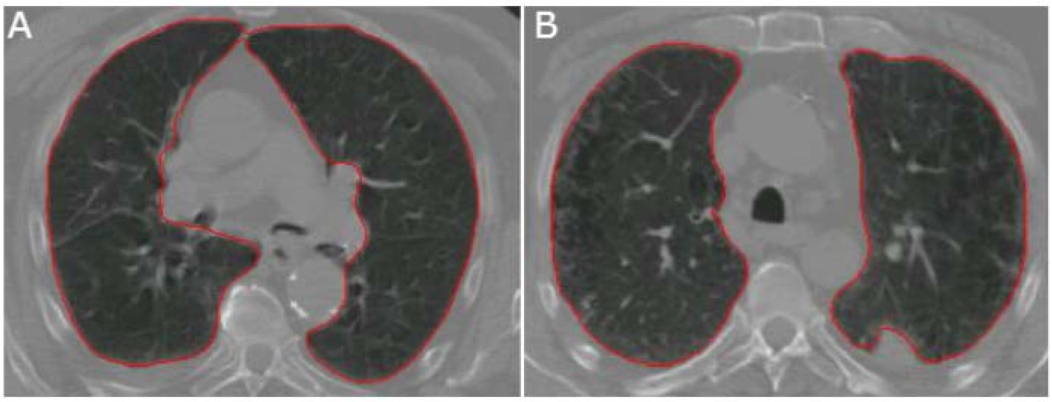


Figure 2.1: Lung CT scan of two patients. Patient A suffered no side-effect after lung radiotherapy treatment whilst patient B suffered lung pneumonitis. Image from B. Nailon *et al.* [7].

The purpose of the project at the Western General Hospital, is to devise a predictive scheme to determine whether radiotherapy patients will suffer radiation induced-lung fibrosis, or properly called radiation-induced pneumonitis, which can affect the performance of the lung organs due to the related damage to surrounding healthy tissue from the radiation used in the treatment, usually to patients with advanced stages of lung cancer. This prediction is done on patient scans from a CT scanner where an image dataset is constructed from many image planes to form a 3D image. This occurs during the treatment planning stage where a program is devised and constructed for the patient ready to undergo radiotherapy treatment. Past work to predict radiation induced fibrosis has mainly focused on the analysis of 2D images [10]. The novelty of this research is the extension of the analysis into three dimension.

2.1.1 Application overview

In essence, the analysis involves mathematical and statistical methods on patient CT images to generate feature values that help determine if such patient will suffer from post-radiation illness. It is known that visual inspection of images is not possible due it indistinguishable features hence the need to perform detailed mathematical analysis on the image, represented in numerical form when treated as input into a processing function in MATLAB. A CT scan pair is shown in Fig. 2.1 from the Hospital's research work where two patient scans look similar yet one suffered from pneumonitis whilst the other did not.

The mathematical features performed on such CT scan images, after conversion to a gray-scale numerical form, are:

- 3D First-Order-Statistics
- 3D GTSDM

- Haralick Features
- 3D GLRLM
- 3D Gabor filtering
- 3D GLSZM

These are techniques to generate texture feature values on CT scan images and are all accumulated into a feature vector of values as a result of applying these algorithms

$$\begin{pmatrix} \text{FOS feature values} \\ \text{GTSDM feature values} \\ \text{GLRLM feature values} \\ \text{Gabor feature values} \\ \text{GLSZM feature values} \end{pmatrix} \tag{2.1}$$

These are then used in a trained classifier system that can ultimately predict whether a patient, from its CT scans, will suffer from post-induced fibrosis based on the generated statistics in Eq. featureMatrix.

2.2 Gabor filtering in medical image processing

From previous preparation work [8], the process of filtering a 3D image using a 3D Gabor filter is the most computational expensive process within the original program, as much as 98% in one case analysed. This section covers the subject of the role of Gabor filtering.

Gabor filtering summarily involves the mathematical convolution of a Gabor filter kernel together with another matrix dataset. In this application example, that dataset is the gray-scale 3D medical CT image scan represented as a double-type 3D array of numbers representing a gray-scale intensity value. Gabor image filtering is known to mimic the perception of what the human eye visualises and the result of images filtered with Gabor filter kernels are highlighted areas that are in the same orientation as the filter kernel wave vector. Examples of image edge detection by convolution of the images with a Gabor filter are demonstrated by a web application by N. Petkov and M.B. Wieling of the University of Groningen [11]. One of their examples is shown here in Fig. 2.2 where the convolution of the left image with a Gabor filter with the wave vector oriented vertically highlights the features in that same orientation.

The medical application here involves the use of Gabor filtering on the gray-scale image in 3D. After subsequent intermediate calculations, including a convolution[2] with the related Gaussian filter, the end result is a single feature value, belonging to a frequency and a unique pair of angles that were used to construct the Gabor filter kernel. With

[2]The MATLAB code uses the correlation, which is equivalent to the convolution with the filter rotated appropriately.

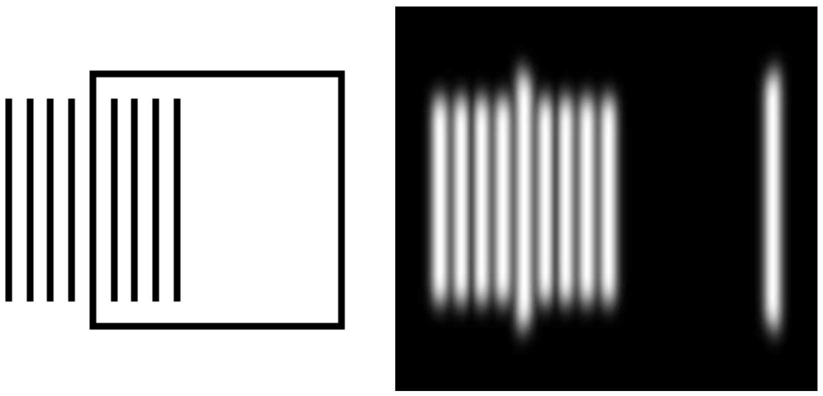


Figure 2.2: Right image is the result of convoluting the left image with a Gabor filter with its wave vector orientated vertically. Image from [11].

a range of frequencies and angles to try, this results in a number of feature values that are part of a wide-ranging features vector as laid out in (2.1). This vector goes on to subsequent processing by a system setup with a classifier scheme to help predict if a patient will suffer post-treatment illness. The raw data used for this is the features vector, including within it the values related to the Gabor filtering calculations, together with patient-specific clinical parameters. This describes one use of Gabor filtering in a medical context, where texture feature statistics generated by applying the filter to gray-scale medical image data are subsequently used to predict the onset of radiation-induced fibrosis in lung cancer patients [7].

Other examples of Gabor filtering applications include its use in liver disease classification [12], for evaluating the performance of an algorithm involved in some aspect of medical image analysis [13] and to track heart images [14].

For the case of liver disease discrimination [12], a similar methodology is employed involving the use of Gabor filters to classify image scans of patient liver images according to the disease type exhibited. The similar problem exists where different liver diseases do not exhibit distinct visual features from patient scans in order for human radiologists to tell the difference between liver disease types and so a more accurate scheme is to extract features from the images using the convolution of those images with a filter bank consisting of a set of Gabor filter parameters to generate a feature vector like the lung project here and use a classifier, trained with sample liver vector data, to classify the images based on the feature values calculated.

Another use of 3D Gabor filtering is its use to evaluate and tune the performance of spatial normalisation algorithms, which are used in quantifying and measuring variance between, for example, a normal image of a brain scan and a noisy scan of the same brain organ [13].

Finally, another example is the use of a Gabor filter bank to track image characteristics of 3D MRI heart patient images [14]. Specifically, 3D Gabor wavelets (in essence Gabor filter kernels with particular frequency, orientation and bandwidth parameters) are used to track lines or planes in 3D heart images scanned using MRI.

Hence in summary, medical imaging analysis involving Gabor filters involve two main stages to extract image texture features from such images:

- Convolute the medical image with a set of Gabor filters of differing frequencies, angles and bandwidth.
- Sum up the energy value matrix derived from the result of the convolution, which is the output/response of the Gabor filter with its wavelet in a particular frequency, orientation and bandwidth. This single sum is the local feature value of the image.

A Gabor filter bank will provide a feature vector consisting of values, each value corresponding to a particular combination of frequency, orientation and bandwidth in the bank. This is the general scheme employed in this lung-pneumonitis prediction application described here and the liver disease classification study [12].

In the grand scheme of things, texture feature extraction through convolution with Gabor filter banks constructs one part of the feature vector tied to a medical image scan that is delivered to a classifier, whose purpose is to use its classification scheme to discriminate against different features. These classifiers are referred to as support vector machines (SVM) because they are loaded with training data in the form of sample feature vectors corresponding to different image classifications such as example feature vectors for images known to have that particular feature. In this case a lung that is susceptible to pneumonitis or a liver image with a particular disease will have known examples of feature vectors that are used. It is the responsibility of this SVM to determine if the feature vector of a given patient CT scan is one that will result in one outcome or another. In this case, either pneumonitis will occur or not. In the case of liver disease, this particular liver scan has this particular liver disease (e.g. cyst, hepatoma and cavernous hemangioma). How well these classifiers perform their job are usually measured using the receiver operating characteristic (ROC), where the area under an ROC measures the classifier's accuracy.

2.3 Benefits of application parallelisation to WGH's research

The aim of this project is to accelerate the original MATLAB program through use of the Intel Parallel SDK to speed up the time-to-results. Clearly from previous preparation work, the time taken to calculate the feature values related to the Gabor filtering can immensely benefit from parallel computation as the existing Gabor filtering takes in the order of hours to process, or in the case of the largest data sample provided, even days.

We aim to provide the following benefits to WGH's application code:

- Equivalent functionality of Gabor filtering application through use of the Intel Math Kernel Library equivalent routines.

- Utilise parallel programming techniques to make use of available multi-core resources to achieve speed-up.
- Implementation of Gabor filtering routines onto Intel Xeon CPUs.
- Provide prototype demonstrations on native compiled programs to provide an idea of native performance over MATLAB code.
- Enable the existing MATLAB code to use our developed routines in C through MATLAB's MEX framework.
- Implementation of Gabor filtering routines offloaded onto Intel Xeon Phi co-processors.

The success of our project will be evaluated based on the following criteria:

- Serial performance is better over the original MATLAB application's performance.
- The execution time to obtain the feature values related to the Gabor filtering part exhibits good scalability and use of multi-core resources.
- The answers provided by our solution are correct within an acceptable tolerance.

If these two high-priority criteria can be satisfied within the timeframe of this project, this then provides a good starting stage for future work in this medical application area, such as performance optimisations to reduce time-to-results further.

Our project focuses on acceleration of the Gabor filtering application part of the application with the simple objective of providing the feature values related to Gabor filtering in a faster time. It is beyond the scope of this project to look at the other feature value calculations arising from the other methods such as FOS and GLRLM as previous work has shown that the Gabor filtering application is the most dominant part [8]. This would be left for future work to accelerate the calculation of the other statistical methods if our project results in the time related to the Gabor feature calculations are comparable to the other calculations. An aspect our project does not cover is whether the answers our solution provides a different prediction on whether a particular image will suffer from pneumonitis compared to the original program.

2.4 Chapter summary

An overview of the background research in predicting radiation-induced pneumonitis is provided and an overview of Gabor filtering and its role in medical image processing is provided. Finally, we state the benefits that our project hope to provide by accelerating and parallelising the original MATLAB application provided by Bill Nailon's research team at the Western General Hospital.

Chapter 3

The Intel Parallel Framework, SDK and Hardware Details

In this chapter, the Intel SDK product is introduced together with an overview of the different parallel programming methods available on offer. We also describe the compute hardware available at EPCC that we used these programming tools on. This includes the compute-nodes consisting of Intel Xeon CPUs and the new Intel Xeon Phi co-processors.

3.1 Intel Parallel Studio XE 2013

Intel provides a convenient SDK to take advantage of the advent of multicore CPU hardware in order to leverage the availability of CPU cores. For the foreseeable future, the trend of increasing core count is likely to continue along with continuance of CMOS process shrinkage from current 22nm to 14nm technology and increased power efficiency as demonstrated by consumer reviews of the Haswell-based CPUs[1]. For single or small groups of developers with shared-memory machine architectures, Intel offers a software package suite Intel Parallel Studio 2013, consisting of C/C++ and Fortran compilers, debugging and profiling programs called Inspector and VTune respectively. The alternative package on offer is the Cluster Studio, which is targeted towards larger organisations with distributed systems. Cluster Studio provide Intel's MPI libraries and development headers suitable for these distributed HPC systems. Intel Parallel Studio is the only SDK product that Intel offers under a non-commercial license to developers. For the systems at EPCC, a single seat license for Intel Cluster Studio XE is installed on one of the compute nodes (named *phi*) and has the required support for Intel's Xeon Phi co-processor.

[1]http://www.anandtech.com/show/6355/intels-haswell-architecture, last accessed 21-08-2013.

x86 non-Intel compatibility

The Intel compilers compile code into binaries that run on Intel processors running the x86 instruction set architecture. It is then technically possible to run such generated binaries on AMD processors which are x86 compatible, however they may not run optimally due to differences such as

- Intel SSE4.1/4.2 vs. AMD SSE4a
- Intel Xeon multicores vs. AMD modules with shared FPU

This is made aware to users and developers by Intel's often quoted optimisation notice disclaimer [15]. Although a performance evaluation on AMD systems would be interesting, in this project we only focus on compute-nodes consisting of Intel Xeon CPUs.

3.1.1 Intel Math Kernel Library

For common mathematical processing, Intel offers within the SDK the Intel Math Kernel Library (MKL), containing common functions such as LAPACK for linear algebra, FFT for Fourier Transforms. Another capability offered with MKL is parallelisation of routines by automatic multithreading, hence a developer may choose to develop a single-threaded version of the code and choose automatic parallelisation by compilation with the flag `-mkl=parallel`. For this project involving convolution and correlation, Intel provides functions in their Vector Statistical Library which require the creation of convolution/correlation tasks which are to be executed. In this project, we also explore the MKL API to provide the equivalent functionality as MATLAB's `imfilter` function does in returning the convolution that is the same size as the original image data.

3.1.2 Exploiting single and multicore CPU performance

To extract performance from multicore hardware requires parallel programming methods. There are two ways to make full use of to gain performance from multiple CPU cores:

- **Vectorisation:** Modern CPUs contain vector processing units able to perform arithmetic operations on multiple numbers on the same clock cycle using Single Instruction Multiple Data (SIMD) instructions. For example Streaming SIMD Extension-2 (SSE2) vector units can operate on vectors containing 2 double-precision numbers (128-bit SIMD width as they are known). On Intel Sandy Bridge-based CPUs, the width was increased to 256-bits so that these vectors operated on can store upto 4 double-precision values. On the Intel Xeon Phi co-processor, their performance gains are through their 512-bit vector units by processing vectors containing upto 8 double values.

Before attempting to speed up applications through parallelisation, it is desirable that code is utilising these SIMD units and being able to parallelise floating-point operations through use of these vector units. If compiled code is not vectorised, an order of magnitude of performance is lost when running on a single CPU core. This becomes important when attempting to accelerate applications on the Xeon Phi as a single Xeon Phi core has a slower clock frequency than an Intel Xeon and only 1/8 of the potential processing speed is used as not all available lanes on the 512-bit vector unit are utilised.

- **Parallelisation (on shared memory systems):** The second part of application performance is speedup gained through application multithreading so that many CPU cores can be utilised. The Intel SDK offers a few methods to achieve this, though this is not exclusive to Intel compilers:

  - **OpenMP** This is a standard parallel programming API primarily based on the fork-join parallel paradigm. The methodology is based on the addition of compiler directives therefore it has the advantage of easily maintaining a single source file for both single and multithreaded application with a compiler switch that either processes the pragma directive statements or treats them as comments. OpenMP is at version 4 as of writing [16].

    This is the parallelisation mechanism used in this project as it suits the C language we use here. The parallelisation strategies we employ in this project, based on the fork-join model, make OpenMP a suitable choice for us to multi-thread our application.

  - **Threading Building Blocks** This is based on a C++ template library where a developer extends one of these templates to code what a thread should do with a given piece of data. An example of its use is in the calculation of the Mandelbrot set for different areas of the complex plane where a task scheduler distributes blocks of the plane to different running threads. This parallel paradigm appears to be better suited to task based parallelism where, for example, distinct tasks based on the different domain chunks distributed are taken and worked on. The Threading Building Blocks (TBB) parallel paradigm is supported by the Intel compilers through the compile flag `-tbb` though it is not an Intel exclusive method and the technology is open-source and functional on 3rd party compilers[2]. It is even possible to use it for Android smartphone application development. Intel maintains commercial support for TBB whilst open source versions are available for anyone to use freely.

    Because the primary language for project was decided to be C, this C++ based method was discounted.

  - **Cilk Plus** This provides language extensions to C and C++ that are suitable for applications that use data decomposition or taskfarm-based parallel

[2]Intel TBB website - http://threadingbuildingblocks.org, last accessed on 21-08-2013.

methods[3]. One of the attractive features is its array syntax similar to array notation in Fortran or MATLAB, allowing developers to easily express vector operations on data that the compiler can map onto SIMD vector units.

Because the computational routines is to be provided by Intel MKL library functions, we do not see using Cilk Plus as suitable for this project.

3.2 Hardware

On the hardware side, we have the availability of multicore CPUs on multisocket systems providing a capable hardware platform for HPC applications at the node-level. The scope of this project is to focus the application at the node-level and not consider distributed memory systems such as connected cluster implementations. At EPCC there are two nodes, named *fermi-0* and *phi*, that were mainly used for this project, both dual-socket Intel Xeon systems with large memory banks. *fermi-0* has MATLAB available and was extensively used to prototype, test and verify our application with WGH's MATLAB code. The *phi* node is equipped with newer Sandy Bridge-based Intel Xeons, more available memory and two Intel Xeon Phi co-processor cards[4] connected using PCIe.

3.2.1 Specifications

Table 3.1 details the hardware specifications[5] of the *fermi-0* and *phi* compute nodes.

	fermi-0	*phi*
CPUs	Two Intel Xeon X5650	Two Intel Xeon E5-2650
CPU Frequency (TurboBoost)	2.67GHz (3.06GHz)	2.00GHz (2.80GHz)
Core count	12 (6 cores each)	16 (8 cores each)
Thread count	24 (2 per core)	16* (1 per core)
Level 3 cache	12MB	20MB
Single Thread performance	1,357 marks	1,192 marks
Max Memory Bandwidth	32GB/s	51.2 GB/s
Total Node Memory Capacity	48GB	64GB

Table 3.1: Hardware specifications of the compute-nodes.

*Although the Intel product website states that each core runs two threads, it appears disabled on the *phi* node. This is confirmed by the relevant output near the end:

[3]Intel Cilk Plus website - http://www.cilkplus.org, last accessed on 21-08-2013.

[4]Product Model 5110P.

[5]Sources: Intel Product Information website (http://ark.intel.com), CPU Benchmarks by PassMark Software (http://www.cpubenchmark.net)

```
On fermi-0 node:

$ cat /proc/cpuinfo
  ....
processor     : 23
  ....
physical id   : 1
siblings      : 12
core id       : 10
cpu cores     : 6
  ....
  ....
```

```
On phi node:

$ cat /proc/cpuinfo
  ....
processor     : 15
  ....
physical id   : 1
siblings      : 8
core id       : 7
cpu cores     : 8
  ....
  ....
```

where we see that on the second CPU (phy. id = 1), there are 8 cores but only 8 threads (siblings = 8). On the *fermi-0* node where each core runs 2 threads, the siblings value is twice that of CPU cores.

3.2.2 The Intel Xeon Phi co-processor

This is Intel's competitive product aimed for high performance and scientific computing based on their Many-Integrated-Core (MIC) processor architecture[6]. The form factor we use is the 5110P model (Fig. 3.1) with the Intel Xeon Phi co-processor, consisting of 60 cores, paired with 8GB of GDDR5 on-board memory. Although this can fit into a board with a PCIe slot, this cannot be slotted into a desktop PC due the lack of an active fan and the board needs to satisfy compatibility requirement, as we found out when attempting to install the product onto an AMD system.

Intel's product provides upto 240 threads of execution from a single chip. The thread count can be increased further by installation of multiple Xeon Phi cards on a single node. Parallel applications are expected to use all available threads in order to hide memory access latencies inherent in an in-order core design.

Intel labels the Xeon Phi as a co-processor due to the capability to share some of all of the parallel workload of the same executing program across Xeons and Xeon Phis installed on a node. This can be achieved either by offload compiler directives or by using MPI with processes on both the node frontend (CPUs) and the backend (Xeon Phis). In either case, Intel's proposition allows the software development process to maintain a single version of the program source code that is able to be compiled for different targets as schematically shown in Fig. 3.2.

Technically, the Intel Xeon Phi is seen as 240 virtual x86 CPU cores on a single PC hence the term many-core. The thinking is that any parallel applications optimised that scales well on multicore CPU nodes should in theory also scale well for 240 virtual cores and many more.

[6]http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html

WARNING: REMOVE ALL PROTECTIVE COVERS PRIOR TO INSTALLING INTO SYSTEM
intel
Xeon Phi™ Coprocessor

Figure 3.1: The Intel Xeon Phi 5110P product installed on *phi* at EPCC.

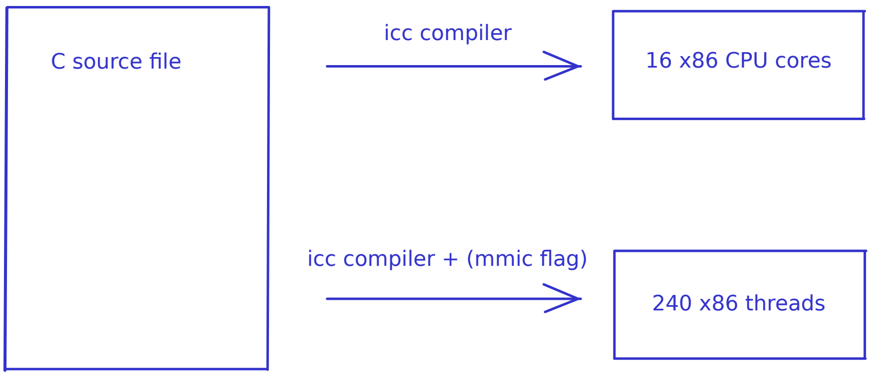


Figure 3.2: The software development process enabling the same code to target different CPU hardware.

Each Xeon Phi core is a P54C revision of the original Pentium core with three important additions:

- **A 512-bit SIMD vector unit.** The Xeon Phi attains its performance by being able to process vector calculations with a vector unit that is able to hold 8 double-precision numbers. To achieve high performance, an application must utilise these vector units. Additionally, the Xeon Phi cores are capable of fused-multiply-add (FMA) operations in the same clock cycle. Hence the quoted peak performance of approximately 1TFlop double-precision from Intel's product pages is derived from:

$$(1052.63\,\text{MHz}) \times (60\text{ cores}) \times (8\text{ SIMD-DP ops} \times 2\text{ FMA /cycle}) \tag{3.1}$$

- **64-bit addressing capability.** The old Pentium core which originally had 32-bit has been upgraded to 64-bit to address more than 4GB of memory, making it more suitable to large-scale scientific and numerical applications.

- **Multi-threading.** Each core runs 4 virtual threads. Its purpose is to hide the access latencies inherent in an in-order CPU design so that the core's scheduler can switch amongst the threads to keep the core busy with work. It is important that parallel applications make use of all possible threads. Furthermore, running a single thread on each core would only utilise half of the core's maximum Flop performance [17]. Due to the intended use of the hardware i.e. running parallel applications using all possible threads, the core's scheduler expects to switch amongst more than one running thread. With only one thread to choose, every other clock cycle is wasted due to the scheduler unable to switch to a second running thread. Therefore, applications must expose enough concurrency to run on at least two threads per core, making a total of at least 120 threads in order to possible make full use of the core's capabilities.

Figure 3.3[7] shows the architectural similarities between Intel Xeons and Intel Xeon Phis when viewed as a node-level system. Some interesting comparisons between a traditional x86-SMP node of multicore CPUs and the 240 virtual cores on the PCIe card:

- **Cache coherency mechanism.** Both x86 CPUs and the cores on the Xeon Phi contain L1 and L2 caches, with Intel Xeons extending to a larger L3 cache. Both types of processors maintain cache-coherency amongst all available cores under a variant of the MESI protocol [18]. Whilst x86 CPUs tend to use a snooping based protocol, the Xeon Phi uses a directory-based protocol consisting of distributed tag directories that are evenly spread along the ring bus interconnect.

- **Memory access** On CPU nodes, each CPU has an on-die memory controller which links the CPU to main memory. For the Xeon Phi, all 60 cores are linked

[7]Intel Xeon diagram is based on the Sandy Bridge EP block diagram - https://computing.llnl.gov/tutorials/linux_clusters/images/sandyBridgeEP.400pix.jpg, last accessed 21-08-2013.

using a bi-directional ring-bus with GDDR5 memory controllers at regular locations on the ring for data transfers from main memory to the core's cache. In the Intel Xeon Phi's case, memory access is symmetric whilst for multisocket systems, each CPU is directly connected to separate memory banks in a cc-NUMA fashion.

- **Operating System** Both CPU types are capable of running an operating system. Whilst x86 nodes typically run an off-the-shelf GNU/Linux, the Xeon Phi runs a customised version of GNU/Linux named Manycore Platform Software Stack[8] (MPSS). Access to a command-line on the Xeon Phi is achieved by secure-shell connection from the hosting node.

The programmability of the Intel Xeon Phi follows the same process as for targeting a traditional x86-SMP node, making use of vectorisation units and exposing as much parallelism as possible.

In this project, our interest is running our developed code on Intel Xeon Phi hardware after development, testing and extracting parallel performance on a traditional x86-SMP node of Intel Xeons with between 12 and 16 available cores. The Intel MKL library functionality is supported on the Xeon Phi from version v.11 available on the EPCC *phi* node. Developing a working framework where the GNU Octave [19] programming environment offloads work onto the Intel Xeon Phi is explored in addition to running native compiled versions of the existing source code on the Xeon Phi itself. We do not explore work distribution amongst more than one Xeon Phi co-processor.

3.3 Chapter summary

We have introduced the Intel SDK to be used to accelerate and parallelise the medical application, of which the provided C compiler and the Intel Math Kernel Library are to be used. The different parallel methods for shared-memory systems were introduced and that OpenMP was the choice for this project. The hardware details of the two shared-memory nodes to be used were presented. Finally, the Xeon Phi was introduced along with a brief look at the architectural similarities to a dual CPU shared memory system.

[8]Intel Manycore Platform Software Stack (MPSS) - http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss, last accessed 21-08-2013.

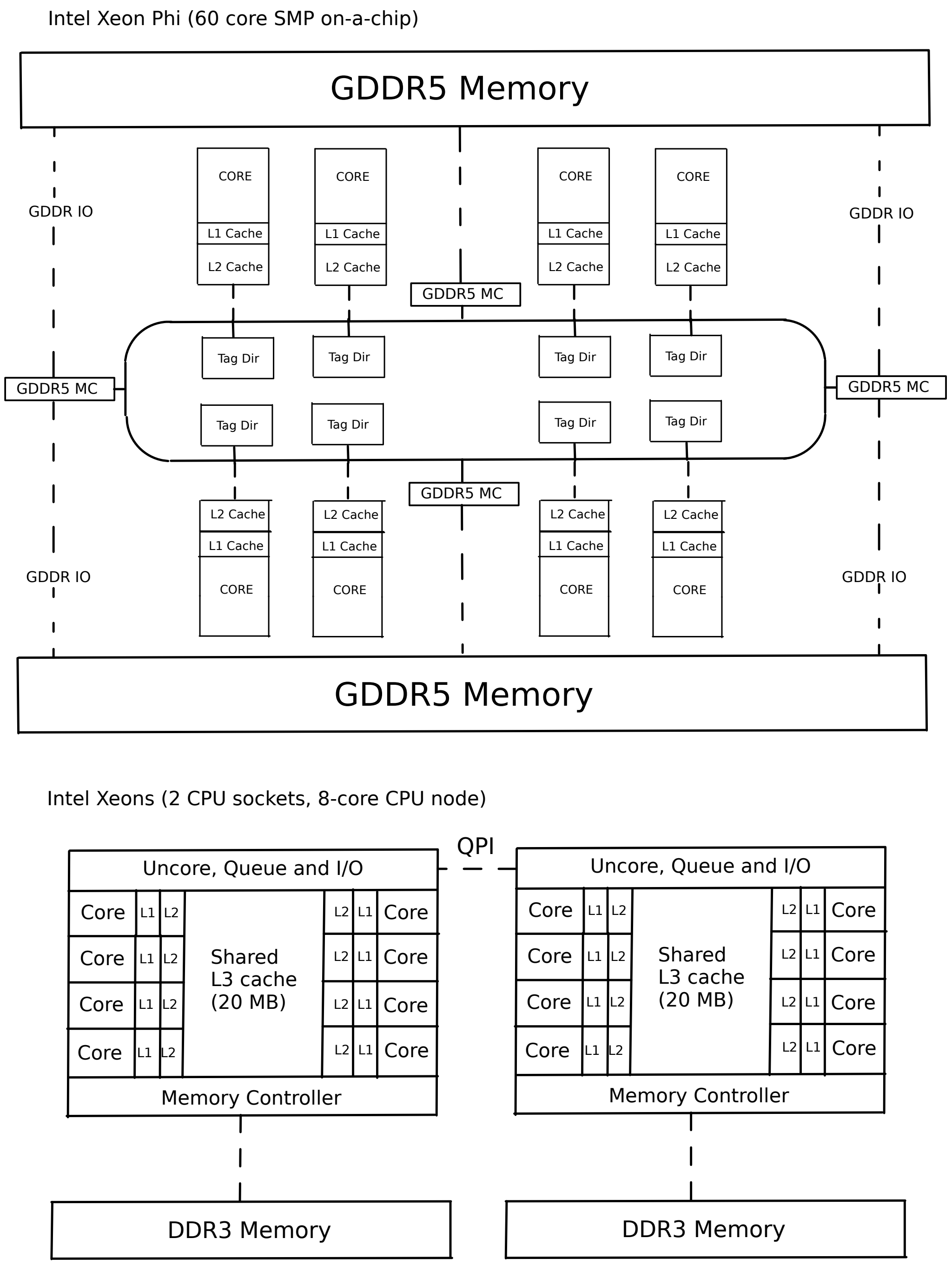


Figure 3.3: Architectural similarities between an Intel Xeon system and an Intel Xeon Phi product. Images based on Xeon and Xeon Phi [20] architecture diagrams.

Chapter 4

Convolution/Correlation: Implementation Details and Parallelisation

This chapter covers the convolution and correlation algorithm that is used in the most expensive part of the medical application program. The two methods in which it can be computed, either directly or through use of Fourier Transforms, is mentioned. The next sections show the implementation of these algorithms using the MKL library from the Intel SDK and to provide equivalent functionality of the implementation used in MATLAB. This includes a brief overview of interfacing native C programs with the MATLAB computing environment. Finally we describe the two main parallelisation strategies used in this project.

4.1 The Convolution/Correlation

The three-dimensional discrete convolution of a matrix $\mathbf{A}$ with another, $\mathbf{B}$, is calculated from the mathematical formula[1] (4.1).

$$\mathbf{C}(i,j,k) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} \sum_{p=-\infty}^{\infty} \mathbf{A}(m,n,p)\mathbf{B}(i-m,j-n,k-p) \tag{4.1}$$

Matrix $\mathbf{A}$ is labelled as the data and $\mathbf{B}$ as the filter kernel. The correlation is a similar calculation[2] for two matrices (4.2)

$$\mathbf{C}_{\text{corr}}(i,j,k) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} \sum_{p=-\infty}^{\infty} \mathbf{A}(m,n,p)\mathbf{B}(m+i,n+j,p+k) \tag{4.2}$$

[1]Using MATLAB's definition - http://www.mathworks.co.uk/help/matlab/ref/conv2.html, last accessed 16-08-2013

[2]We assume the data is real-valued.

where one needs to transform the filter kernel $\mathbf{B}$ appropriately and apply the same convolution formula to obtain the correlation. Hence for much of this chapter, we cover the details of the convolution which are similar for the correlation.

The indices are practically taken to be finite and the truncation point will depend on the type of convolution desired. Suppose the data $\mathbf{A}$ is of size (N_1, N_2, N_3) and the filter $\mathbf{B}$ has size (K_1, K_2, K_3). Then the two types of convolution we are interested in are:

- **Full convolution**: the matrix size of $\mathbf{C}$ is $(N_1+K_1-1, N_2+K_2-1, N_3+K_3-1)$.
- **Same convolution**: the central part of the full convolution is returned for $\mathbf{C}$ and is the same size as the input data $\mathbf{A}$. The start of the central convolution is determined by the size of the filter. However this starting point can be chosen arbitrarily and both MATLAB and GNU Octave (v3.6.4) start at indices

  $$(\lfloor K_1/2 \rfloor, \lfloor K_2/2 \rfloor, \lfloor K_3/2 \rfloor) + (1, 1, 1) \tag{4.3}$$

  of the full convolution.

  In both cases, data points that fall outside the domain of $\mathbf{A}$ are assumed to be zero.

The convolution can be thought of as the filter sliding around the input data. Figures 4.1 and 4.2 show this concept in 2D but can easily be visualised for 3 dimensions[3].

Figure 4.1 shows obtaining the full convolution of a matrix $\mathbf{A}$ (data in the figure) of size with $N = 5$ in each dimension and a filter $\mathbf{B}$ of length $K = 3$ likewise with the matrix entries specified in the figure. The convolution is the multiplication of each matrix element that overlap each other and then all summed together into the appropriate entry. So the first matrix index at the top-left of the full convolution, $\mathbf{C}(1, 1)$, only involves one pair of overlapping elements. So $\mathbf{C}(1, 1) = 1 \times 17$. Similarly,

$$\mathbf{C}(1, 7) = 1 \times 15 \tag{4.4}$$
$$\mathbf{C}(2, 2) = -8 \times 17 + 24 + 23 + 5 = -84 \tag{4.5}$$

Figure 4.2 illustrates obtaining the same convolution using the same input data as in Fig. 4.1 but we illustrate with two different sized filters to show how the central convolution is obtained. The starting index for the same convolution can be used to determine the center of the filter. This center point overlaps the first entry of the data and is where the first entry of the convolution is stored.

Figure 4.3 illustrates rotating and reflecting the filter to obtain the correlation by convoluting the data with this transformed filter.

The medical imaging application uses correlation of the 'same' size as the image data. The Intel MKL libraries for computing the convolution/correlation return the full answer but offer a way to set the starting index of the answer. We describe obtaining equivalent functionality between them in section 4.2.1.

[3]These figures are inspired by the diagrams from an Nvidia whitepaper on FFT-based 2D convolutions [21].

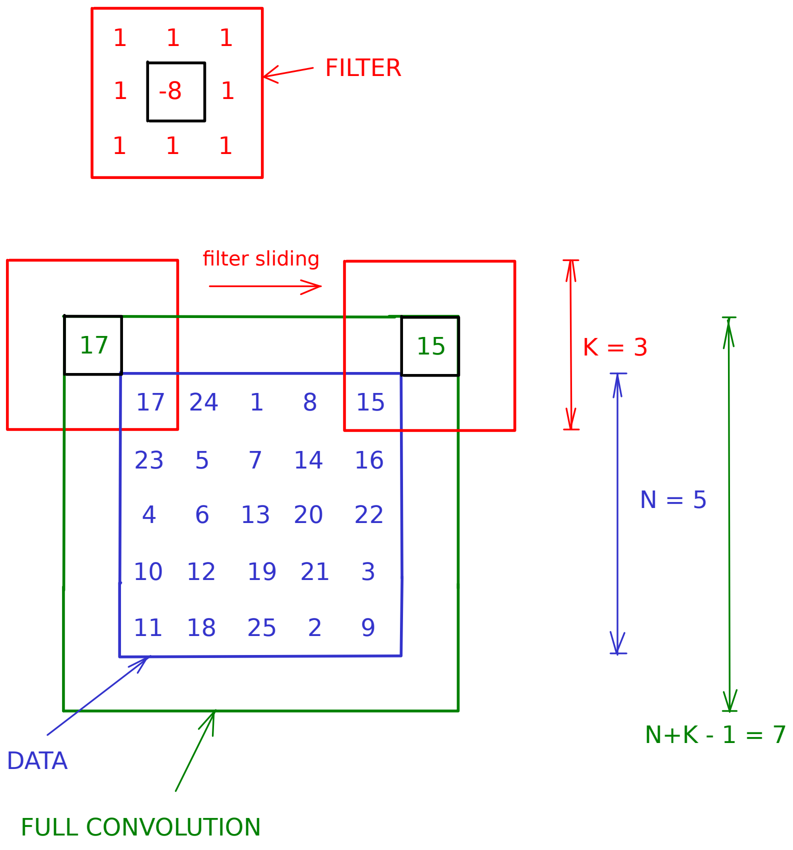


Figure 4.1: Illustration of the full convolution for a data of size (5,5) with a filter of size (3,3). The full convolution size is therefore (7,7).

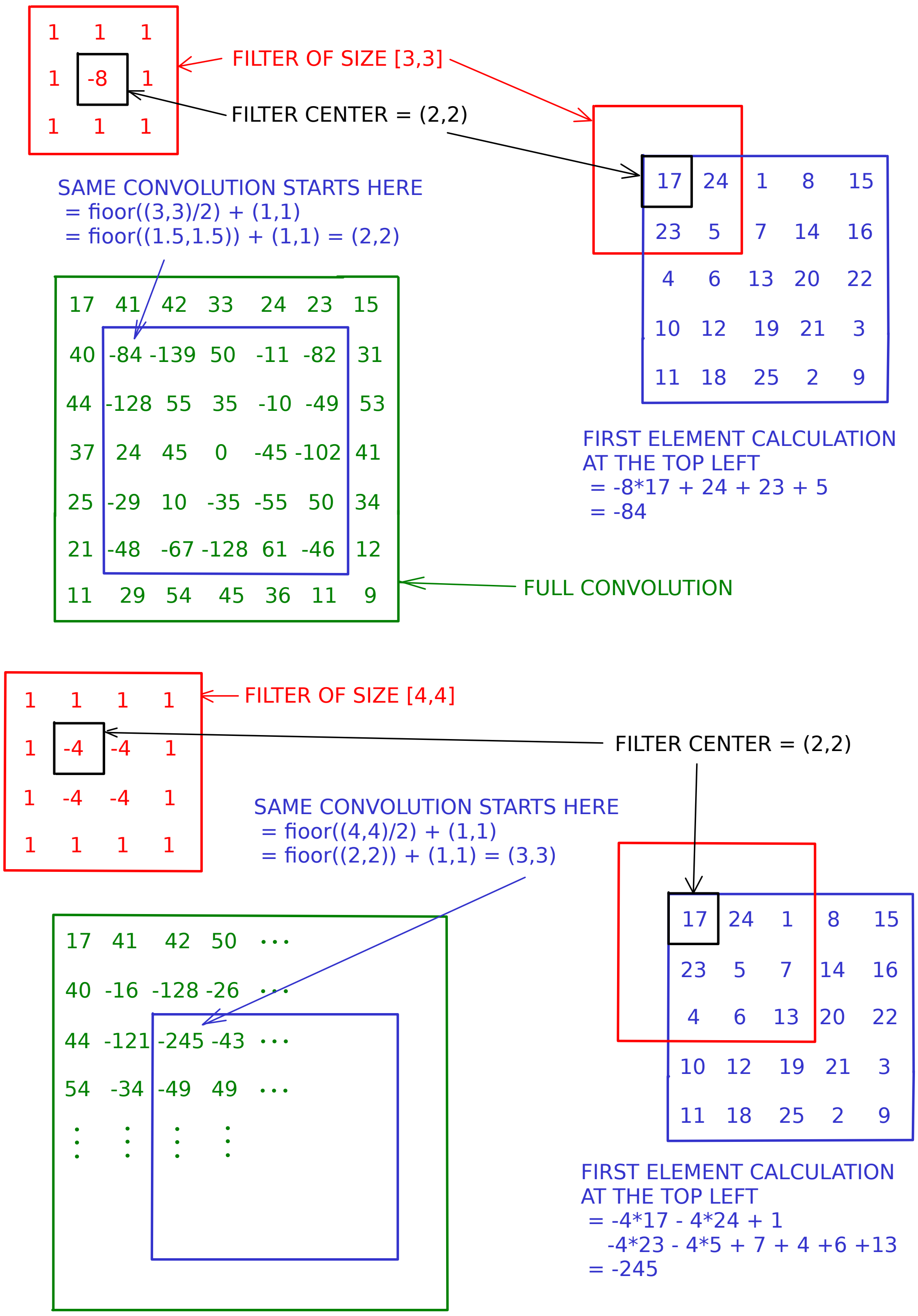


Figure 4.2: Illustration of the same convolution for a data of size (5,5) with filters of size (3,3) and (4,4).

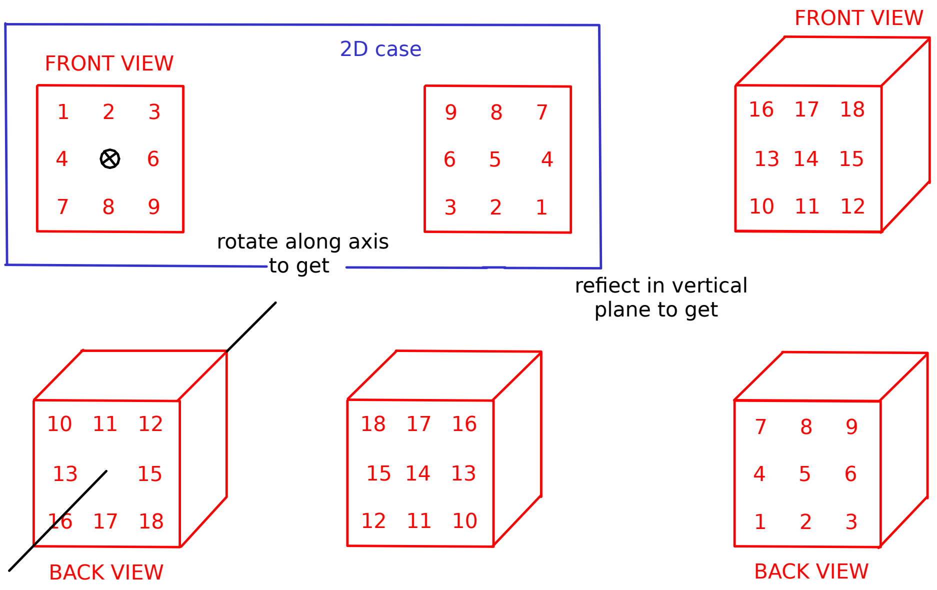


Figure 4.3: Illustration of rotating the filter and applying the same convolution algorithm (4.1) to obtain the correlation in (4.2).

4.1.1 Convolution/Correlation by Fourier Transforms

The use of Fourier transforms can be used to compute the convolution by the circular convolution theorem [22]. Both the data and filter are transformed into new matrices as shown in Fig. 4.4. To obtain the same convolution from the circular convolution, the filter is shifted so that its center corresponds with the first element of the 'same' convolution.

Suppose that $\mathbf{A}$ and $\mathbf{B}$ are the new data structures in Fig. 4.4 on the bottom left and right respectively. Then applying the formula (4.6) returns the circular convolution of the same size where it is assumed that $\mathbf{A}$ and $\mathbf{B}$ are periodic (imagine copies of themselves tiled together end to end in all directions). The formula involves computing the element-wise product of their Fourier transforms followed by an inverse Fourier transform of the result.

$$\mathbf{A} * \mathbf{B} = \mathcal{F}^{-1}(\mathcal{F}(\mathbf{A}) \cdot \mathcal{F}(\mathbf{B})) \tag{4.6}$$

Because of the way the filter kernel was transformed in Fig. 4.4, the 'same' convolution is the answer obtained with the zero padding stripped off.

Because the convolution by Fourier transforms is based on mathematical theorems, this is not an approximation to achieving the convolution. Hence any numerical differences will be due to the limited numerical precision on computers.

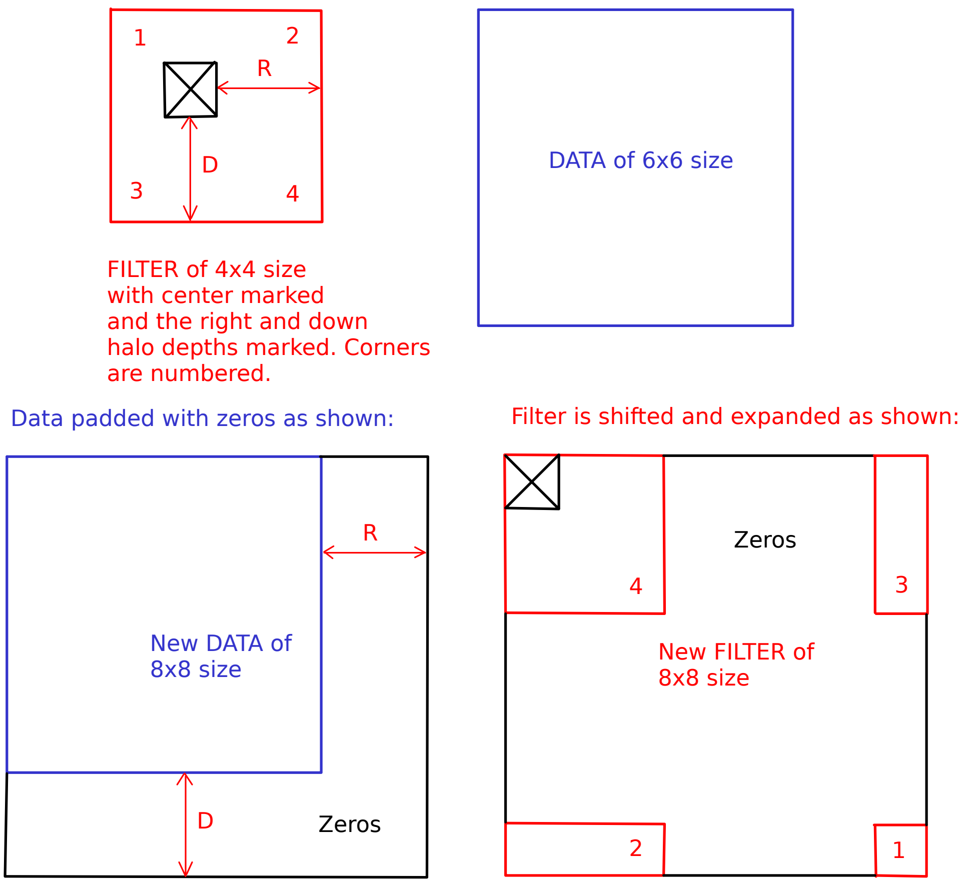

1
2
R
D
3
4
FILTER of 4x4 size with center marked and the right and down halo depths marked. Corners are numbered.
DATA of 6x6 size
Data padded with zeros as shown:
Filter is shifted and expanded as shown:
R
New DATA of 8x8 size
D
Zeros
Zeros
4
3
New FILTER of 8x8 size
2
1

Figure 4.4: Illustration of the required padding of the data and the filter for the purpose of calculating the ‘same’ convolution using the circular convolution theorem, based on [21, Fig.1].

4.1.2 Complexity Comparision

Directly computing (4.1) with an N^3 datasize and a K^3 filter kernel is of order $\mathcal{O}(N^3K^3)$. From our preparation study [8], this can take hours or even days on a single modern processor.

By using the Fourier transform method, one can take advantage of Fast Fourier Transform routines (FFTs) to reduce the computational complexity. For an N^3 datasize and a K^3 filter kernel, typically the padded size of both the data and filter is $P = N + K/2$ in each dimension so the complexity using FFTs is of order $\mathcal{O}(P^3 \log^3 P)$. In addition, the pointwise multiplication of the Fourier transforms adds $\mathcal{O}(P^3)$. The data and filter kernel size need to be sufficiently large such that using FFTs is faster than computing the convolution directly [22, Fig. 7].

4.1.3 The Gabor Filter

The filter used in the original MATLAB code is a Gaussian kernel modulated by a sinusoidal function. We provide the details here because one of our parallelisation strategies will involve creating the filter kernel within the native C program. The formulation is found in other works [12, 13] and we detail what the original MATLAB code uses, which closely follows the formulation by [14].

The Gabor filter h is mathematically defined to be

$$h(\mathbf{x}, \mathbf{x}') = g(\mathbf{x}')s(\mathbf{x}) \tag{4.7}$$

where g is the Gaussian function and s is the modulated sinusoidal envelope function. They are defined as:

$$g(\mathbf{x}') = \frac{\exp(-\frac{1}{2\sigma^2}[x'^2 + y'^2 + z'^2])}{2\pi^{\frac{3}{2}}\sigma^3} \tag{4.8}$$

$$s(\mathbf{x}) = \exp(2\pi i[Ux + Vy + Wz]) \tag{4.9}$$

where $\mathbf{x}'$ is the rotated coordinate system:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = R_z \times R_{xy} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{4.10}$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{pmatrix} \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{4.11}$$

with ψ and ϕ the yaw and roll angles respectively as depicted in Fig. 4.5[4]. The vector $[U, V, W]$ are the components of the frequency F:

$$F = \sqrt{U^2 + V^2 + W^2} \tag{4.12}$$

$$U = V \tan\phi \tag{4.13}$$

$$V = \frac{F}{\sqrt{(1 + \tan^2\phi)(1 + \tan^2\psi)}} \tag{4.14}$$

$$W = \sqrt{U^2 + V^2}\tan\psi \tag{4.15}$$

Some publications use a minus sign in the exponential for s however, it can be proven that both formulations are equivalent and we settle with the formation used in the original MATLAB program. The σ term is defined as

$$\sigma = \sqrt{\frac{1}{2}\ln(2)}\frac{(2^B + 1)}{(2^B - 1)\pi F\lambda} \tag{4.16}$$

[4]A code mistake in the original MATLAB program was spotted where y' was incorrectly defined.

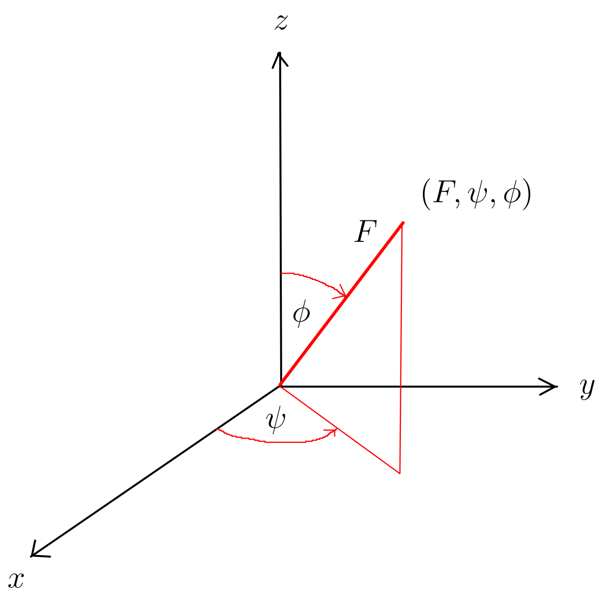


Figure 4.5: The co-ordinate system showing the relation between the yaw (ψ) and roll (ϕ) angles and the frequency F.

which gives a symmetric Gaussian envelope due to the use of a single σ parameter instead of 3 separate ones shown in other publications. B is a bandwidth parameter whilst λ is a parameter taken to be 1 in the code.

In the original MATLAB program, the Gabor filter is created using a discretised mesh of the $\mathbf{x}$ space, going from -3σ to 3σ in each of the 3 coordinates. The number of points is dependent on the frequency F chosen so a higher number of points is taken for higher frequencies to capture the fast moving behaviour of the envelope.

4.2 The VSLConv/VSLCorr routines in Intel MKL

This section presents the relevant MKL routines to perform convolution and correlations. Further details of the API are provided by Intel's MKL manual [23].

The routines for calculating convolutions and correlations are provided by the Vector Statistical Library. The programming involves the creation of what are called task objects which are initialised with optional parameters, such as which algorithm to compute the convolution, and are executed later on, storing the result in pointers to arrays.

A basic skeleton of a typical code for a 3D convolution is as follows:

```
#include <mkl.h>
#include <mkl_vsl.h>
  ...
  int status = 0;
  double *A, *B, *C;
```

```
MKL_INT shapeOfA[3], shapeOfB[3], shapeOfC[3], rank = 3;
...
... // Code to initialise matrices
...
/* Create a convolution task type */
VSLConvTaskPtr convTask;

/* Create a convolution task with specified parameters */
status = vsldConvNewTask(&convTask, algorithm, rank,
                          shapeOfA, shapeOfB, shapeOfC);
(check status)

/* Execute the convolution task */
status = vsldConvExec(convTask, A, NULL, B, NULL, C, NULL);

/* Delete task when done */
status = vslConvDeleteTask(&convTask);
...
```

The main points of the above code is

- A convolution task object is declared and is initialised and created with the task creation routine `vsldConvNewTask` appropriate for double-type matrices. There are separate creation routines for complex number and single precision types.
- Computing the convolution directly or by using FFTs can be set by passing one of these as the `algorithm`:
  - `VSL_CONV_MODE_DIRECT`
  - `VSL_CONV_MODE_FFT`
- The `rank` is set to 3 for a 3D convolution and expects the shape of the pointer arrays to contain 3 entries. For example, a regular 3D array of length n would pass in a 3-length array $[n, n, n]$.
- After creation, the task is executed and returning the answer to the third pointer array `C`. An option exists to set the stride length whilst accessing array elements.
- Specifying the shape of the array `C` to be $[N+K-1, N+K-1, N+K-1]$ for an array `A` of size N^3 and filter `B` of size K^3 returns the full convolution in `C`.
- The task object should be deleted to free up any internal memory after use.
- The status should be checked each time for any error codes returned.

To define a correlation task instead, the names used in the code snippet above simply substitute instances of `Conv` with `Corr`. The following lines return equivalent answers for array `C`:

```
vsldConvExec(convTask, A,  NULL, B, NULL, C, NULL);
vsldCorrExec(corrTask, Br, NULL, A, NULL, C, NULL);
```

where `Br` is the rotated and reflected filter kernel of `B` as shown in section 4.1.

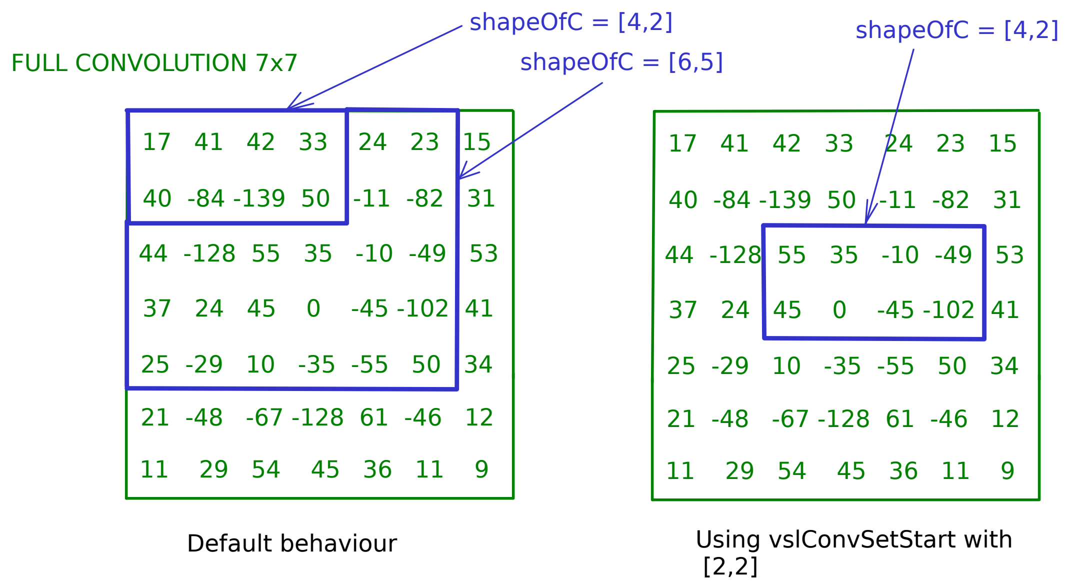


Figure 4.6: Shows the effect of specifying an offset of [2,2] and not specifying an offset at all. The 2D case is illustrated.

4.2.1 Obtaining the ‘same’ convolution

Obtaining the ‘same’ convolution from the full convolution that is returned by the task object requires us to set the starting index of the convolution answer in the task object. Figure 4.6 illustrates the convolution task object’s default behaviour on the left-hand side. So specifying the shape of the answer returns the convolution of the specified size but starting at the top-left point of the full convolution.

To specify the same central convolution that MATLAB returns, we can set the starting point where the convolution task object starts in the full convolution. The code to achieve this is:

```
MKL_INT offset[3];
  ... // Specify the offsets
status = vslConvSetStart(corrTask, offset);
```

in the 3D case. Figure 4.6 shows the effect, in 2D, of passing [2,2] as the offset so that the starting point is now 2 entries further down the full convolution answer. The offset is determined from the size of the filter kernel in the same was as MATLAB does for the starting point of the central convolution. The shape of the answer `C` is simply the same as the shape of the original matrix `A`.

For correlation tasks, the offset is specified differently in order to return the ‘same’ convolution because the matrices are swapped in the function arguments. The offset how far off from the filter center we want to start the answer from. In the project source code, it is calculated to be

$$\text{offset}[i] = K_i/2 + 1 - K_i;$$

in each of the three dimensions i.

The native C codes found in the source code directory for this project that perform the correlation and convolution are

- imfilter3D_MKL.c: convolution code using VSLConv functions, equivalent to MATLAB's imfilter with the 'conv' option.
- imfilter3Dcorr_MKL.c: correlation code using VSLCorr functions, equivalent to MATLAB's imfilter.

4.2.2 Task object reuse

The MKL library offers versions of the convolution and correlation routines that allow reuse of the task objects. For example, a task object can be used with different filters on the same input data. This potentially offers advantages where much of the internal setup for that same input data has been done and the execution can potentially reuse any intermediate calculations. This is especially useful when the mode of operation is FFT mode where the Fourier transform of the input data does not have to be computed on every execution if the same task object can be reused.

The difference in the skeleton code compared to the previous one is

```
...
...

/* Create a reusable convolution task with specified parameters */
status = vsldConvNewTaskX(&convTask, algorithm, rank,
                          shapeOfA, shapeOfB, shapeOfC,
                          A, NULL);

/* Execute the convolution task */
status = vsldConvExecX(convTask, B, NULL, C, NULL);

...
...
```

The function is named with an X-suffix and the input data array `A` is now part of the task creation process. Upon execution, this can be invoked many times with different filter kernels `B`.

We use this in one of our parallelisation strategies where different Gabor filter parameters generate different filter kernels but operate on the same image data. Hence we hope to gain additional performance through this strategy of task object reuse.

4.3 MATLAB MEX Framework

Using the Intel SDK requires our code to be C-based, whereas the original code provided by Western General Hospital runs under the MATLAB numerical programming environment. To provide MATLAB with access to these C-based routines, MATLAB provides an API allowing the MATLAB runtime environment to call compiled C code, namely the MEX framework [24].

A C program requires a function named `mexFunction` that will be called by the MATLAB. The template body is

```
#include <mex.h>

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
  ...
}
```

where MATLAB passes any function arguments provided by the user into the mexFunction using the `prhs` pointer array. The C code has the option to return results back into MATLAB using the `plhs` pointer array.

To compile with the Intel compilers, the following command line and associated compile flags are used:

```
   // Compiling source code
   icc -I/usr/local/MATLAB/R2012a/extern/include -fPIC -O3 -openmp \
                           -mkl=sequential -c -o object.o source.c

   // Linking to create MEX binary
   icpc -mkl=sequential -lmkl_rt -shared -o myProgram.mex object.o
```

Highlights of the above commands:

- The extension is `.mex` or `.mexa64` for 64-bit systems such as *fermi-0*.
- The include files are the MATLAB header files located in the MATLAB installation directory.
- Code must be compiled as a Position-Independent-Executable. MATLAB complains and refuses to compile otherwise.
- Linking with MKL runtime library by the `-lmkl_rt` flag allows use of the MKL functions. The `-mkl` flag can either use the sequential or parallel version of the MKL libraries.
- The shared flag is necessary for some static linking necessary in order for MATLAB to correctly identify and invoke the routines from the MKL libraries. Without this flag can result in a segmentation fault and abnormal termination of the MATLAB runtime environment.

- The ability to thread an application with OpenMP for MATLAB to use is possible.

We conclude that we can successfully link developed C code, compiled using Intel compilers, with the MATLAB runtime environment. This is despite no stated support for such compilers by MathWorks Inc for Linux[5].

4.3.1 GNU Octave

MATLAB is a commercial product that requires valid licenses to run and operate legally. To simulate the use of MATLAB on the *phi* compute-node with faster Intel Xeon CPUs and the Intel Xeon Phi co-processors, we use the Octave[6] programming environment that is mostly MATLAB-syntax compatible [19]. This allows us to test interfacing with the Intel Xeon Phis available on the *phi* node whereas they are not available on the *fermi-0* node where MATLAB is installed.

The method to generating MEX files that Octave can use is the same method as described previously for MATLAB. The differences are

- The header files are located in the Octave include directory e.g.

  `/usr/include/octave-3.4.3/octave`

  on the *phi* node.
- The `-lmkl_avx` link flag is required for Octave in order for MKL to function whilst using the AVX vector units on the CPUs.

4.4 Parallelisation

The section will look at two main parallelisation strategies using OpenMP. The first one involves distributing the data to be convoluted/correlated into subdomains on different threads. The second distributes the Gabor filter parameters and each thread calculates a feature value. This involves convoluting the entire data with the particular Gabor filter generated by those parameters.

4.5 Parallelisation by Domain Decomposition of Input Data

The convolution algorithm is highly parallelisable in terms of independent domain decomposition where no boundary data needs to be communicated amongst threads during

[5]Supported and Compatible Compilers - Release 2013a - http://www.mathworks.co.uk/support/ compilers/R2013a/index.html?sec=glnxa64

[6]The full name is GNU Octave but we refer to it as Octave for convenience.

computation.

This parallel strategy allows us to provide a replacement function for the MATLAB `imfilter` routine with an equivalent one that distributes the data onto multiple threads. Hence this strategy is 'function substitution' and the function takes the input data, filter and the number of threads to use to perform the convolution.

4.5.1 Parallel Strategy Outline

An overview of the parallel algorithm is described below and Fig. 4.8 shows an example of the strategy using four threads on a 9-by-9 dataset with a 4-by-4 filter.

Step 1: Arrange Threads into a 3D Grid

The first step is to arrange the running OpenMP threads into a 3D Cartesian grid in a dynamic fashion. This allows individual threads, knowing where they are in the grid, to correctly copy out the relevant part of the data to be worked on.

There exists such a function in MPI implementations that can automatically create a 3D Cartesian Grid, given a total number of threads running as an input parameter. It can also produce grids automatically if supplied with a predetermined dimension in the supplied output grid array. We use the source code of this thread arrangement functionality from the OpenMPI project [25] and modified to adapt to our multi-threaded implementation. The source code is re-distributable in source and binary forms under the New BSD license hence we are permitted to use and modify the code freely for our project[7].

Step 2: Divide the domain into independent subdomains

After forming the threads into a Cartesian Grid, the relevant portion of the input data is divided amongst the threads according to their position in the grid and copied into the thread's private memory.

We have developed over the course of the project two dividing strategies for dealing with cases where the data length does not divide equally amongst the threads. Figure 4.7 shows the two methods used for an example of dividing data of length 13 amongst 4 threads. We first developed the method of on the bottom that over-divides such that the last thread receives the remainder which is less than the other. However this results in situations where this thread can receive no work, thus under-utilising the number of threads available. Hence the method shown on the top under-divides and the last thread will receive the remainder on top of its existing work.

[7]The source code in our project source code directory is called *dims_create.c*, downloaded from http://svn.open-mpi.org/svn/ompi/branches/v1.7/ompi/mpi/c/dims_create.c

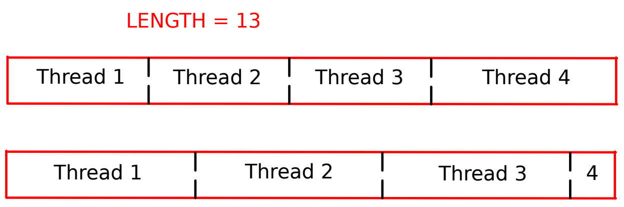


Figure 4.7: Illustration of the division of a 13-length dataset onto 4 threads in two different ways.

To ensure that the convolution is computed corrected when all subdomains are combined together, each thread must also carve out relevant halo data around its own input data, depending on the size of the filter kernel. If the subdomain is on the boundary, the MKL routine assumes a zero boundary and no halo data needs to be appended on such boundary.

Step 3: Each Thread Performs The Convolution

Each thread hold a private copy of the filter and performs the same convolution on their respective subdomains, making use of any halo data when the filter moves outside the subdomain. The code that each thread executes follows the outline code in section 4.2 but using correlation tasks instead to match the original hospital code performing the correlation in MATLAB. Each thread finally writes to a shared array containing the constructed 'same' convolution for the whole data. Knowing where to write in the shared array is determined using the same method to work out which part of input data to copy from.

4.5.2 OpenMP

OpenMP is used to generate the multiple threads that work on independent subdomains of the data to be convoluted or correlated.

The pseudo-code outline is as follows:

```
#pragma omp parallel default(none) \
       shared data(Input Data, Filter, Convolution Answer,
                   vector-tuples indicating the array shapes,
                   Dimensions of Cartesian grid for threads)
{
    // Each thread works out its position in the Cartesian grid
```

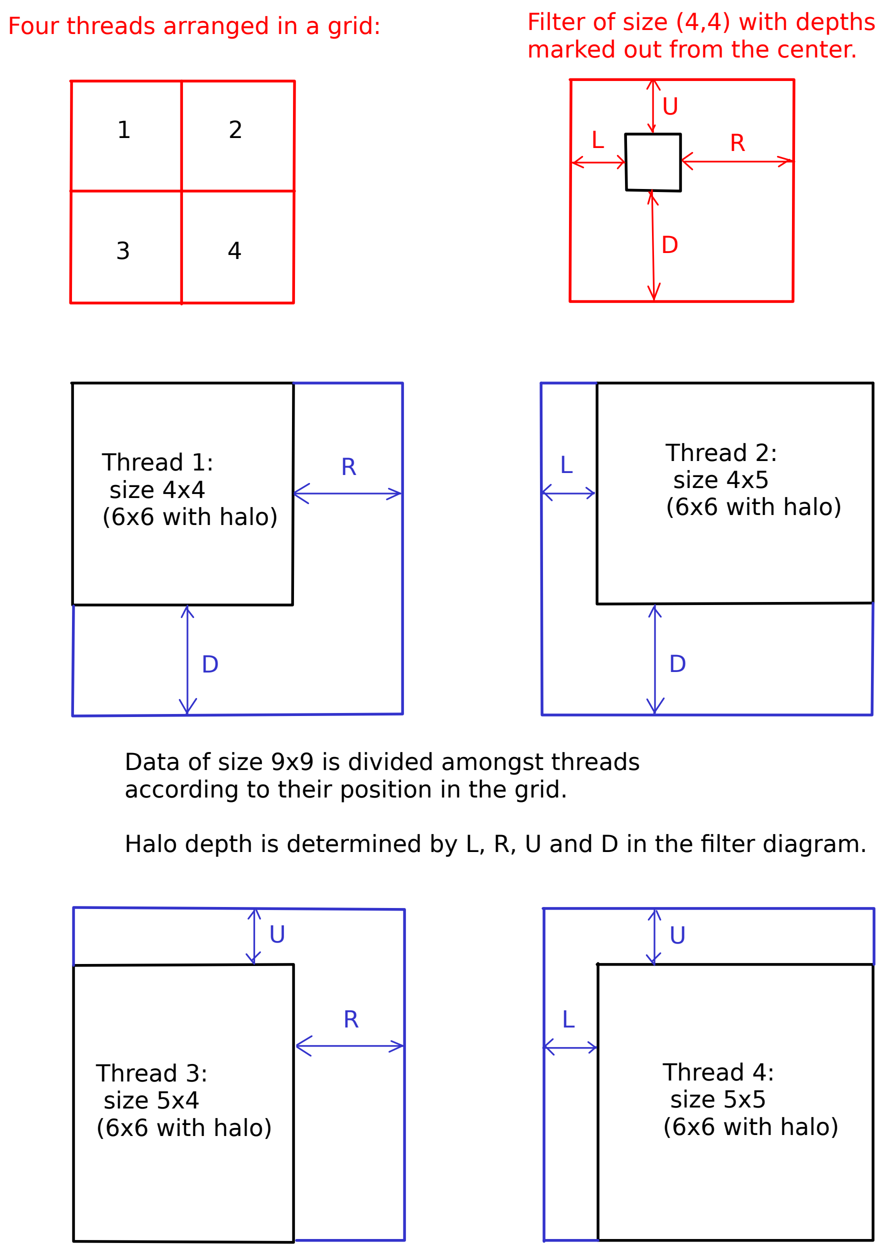

Four threads arranged in a grid:
1
2
3
4
Filter of size (4,4) with depths marked out from the center.
U
L
R
D
Thread 1: size 4x4 (6x6 with halo)
R
D
L
Thread 2: size 4x5 (6x6 with halo)
D
Data of size 9x9 is divided amongst threads according to their position in the grid.
Halo depth is determined by L, R, U and D in the filter diagram.
U
R
Thread 3: size 5x4 (6x6 with halo)
U
L
Thread 4: size 5x5 (6x6 with halo)

Figure 4.8: Example of the division of a 9-by-9 dataset amongst 4 threads and convoluting with a 4-by-4 filter.

```
            ...
    // Each thread allocates private memory space using MKL malloc
          routine for its subdomain, for a private copy of filter
          and for the 'same' convolution of its subdomain.
            ...
    // Each thread copies out the relevant subdomain of the data
       according to their position in the Cartesian grid
            ...
    // Each thread creates, sets up and executes VSLConv task objects
       for their subdomain with the filter
            ...
    // Each thread copies their private answer to the shared
       Convolution answer in the correct place
            ...
    // Clean up code such as freeing arrays and VSLConv task objects
}
```

Thread Safety and Error checking

Each thread in the parallel loop allocates memory using the MKL allocation functions and each creates, executes and deletes the convolution task objects. The test framework developed and the accuracy of the answers returned indicates that the Intel MKL functions are thread-safe. This is confirmed on Intel's website that MKL is designed and compiled for thread safety and can be used in a threaded application[8].

Because OpenMP does not allow individual threads to return from the function, each stage is embodied in an if-else statement so that the procedure terminates if one of the task objects returns an error status.

4.5.3 Use in MATLAB

This parallel strategy enables substituting the correlation routine in the original MATLAB code with our multithreaded C-based routine utilising OpenMP and the MKL correlation routines.

Every instance of MATLAB's `imfilter` routine is replaced by our own developed function that uses the MEX framework in order to call the relevant C code. The following MATLAB code shows the function replacement in the main loop body of the original program responsible for calculating the Gabor feature vector:

```
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);
```

[8]http://software.intel.com/en-us/articles/intel-math-kernel-library-intel-mkl-using-intel-mkl-with-threaded-applications

```
        rh = real(h);
        ih = imag(h);
        kc = my_imfilterCorr_omp(rh,subimg,12); % Function replaced
        ks = my_imfilterCorr_omp(ih,subimg,12); % Function replaced
        E = sqrt(kc.*kc+ks.*ks);
        E = my_imfilterCorr_omp(g,E,12);        % Function replaced
        features(i,j,k) = E(:)'*E(:);

    end
  end
end
```

so the MATLAB runtime calls a compiled MEX file called *my_imfilterCorr_omp*, passing the data, filter and the number of threads to run (12 in the example above). The variables `kc`, `ks` and `E` obtain answers from the parallel C-based routines instead. So we have replaced the original correlation routine with our own that utilises MKL to perform the correlation instead.

4.6 Parallelisation by Task Farming

In the loop body of the original code, there are 4 separate frequencies, 6 yaw angles and 6 roll angles, giving a total combination of 144 feature values computed for each dataset. This taskfarm parallelisation strategy assigns each thread to calculate a feature value associated with that particular parameter combination.

4.6.1 Parallel Strategy Outline

This parallel strategy divides the three nested loops involved:

```
for each frequency value F {
   for each yaw angle value psi {
     for each roll angle value phi {
          ...
         ( calculate correlations and the
           resulting feature value )
          ...
     }
   }
}
```

Each available thread simply processes each iteration and returns the feature value in the appropriate place in the features matrix. This features matrix is of the form

$$\text{features}(:,:,k) = \begin{pmatrix} (F_1,\psi_1,\phi_k) & (F_1,\psi_2,\phi_k) & \cdots & (F_1,\psi_6,\phi_k) \\ (F_2,\psi_1,\phi_k) & (F_2,\psi_2,\phi_k) & \cdots & (F_2,\psi_6,\phi_k) \\ \cdots & \cdots & \cdots & \cdots \\ (F_4,\psi_1,\phi_k) & (F_4,\psi_2,\phi_k) & \cdots & (F_4,\psi_6,\phi_k) \end{pmatrix} \tag{4.17}$$

which is a 3D matrix where Eq. (4.17) shows the k'th 2D plane and the third dimension is for each combination with roll angle ϕ_k.

4.6.2 Serial Code Outline

Provided is an overview in pseudo-code of the native program run by a single thread:

```
void generate_Gabor_features (input data and its shape,
                              filter kernel,
                              Gabor filter parameters [F,phi,psi],
                              features vector)
{
   ...
   For each frequency F {
      ...
      Determine the resolution of the filter kernels
      Generate convolution task objects
      ...
      For each yaw angle psi {
        For each roll angle phi {
          ...
          1. Generate the Gabor and Gaussian filter kernels
              (suitable for convolution)
          2. Execute the convolution task objects with the Gabor filter
          3. Calculate the energy matrix from the real
              and imaginary answers
          4. Convolute the energy matrix with the Gaussian filter
          5. Compute the Gabor feature value from the convoluted
              energy matrix and write into the features vector
        }
      }
   }
   // The feature vector contains all 144 Gabor feature values of
   // the data.
}
```

The above code shows how much of the Gabor feature calculation code is migrated from MATLAB to a native C program. We expect this code to perform faster as a result.

We use the convolution routines instead of correlation because one of the taskfarm parallelisation methods allows us to reuse the task objects for each frequency value rather than creating and destroying it in each loop iteration.

The filter resolution is dependent on the frequency value F. If the frequency is higher, the mesh resolution is finer in order to capture the behaviour. Table 4.1 states the mesh resolution of the filter kernels for the four frequencies used in the original imaging program.

F value	Filter kernel meshsize
25	$14 \times 14 \times 14$
50	$17 \times 17 \times 17$
75	$17 \times 17 \times 17$
100	$24 \times 24 \times 24$

Table 4.1: The filter kernel sizes for the four different frequency values.

4.6.3 OpenMP Parallelisation of Angle Parameters

We use the `for collapse(2)` OpenMP directive on the two innerloops involving the angles, shown in the pseudo-code outline below:

```
 ...
#pragma omp parallel default(none) \
        shared data(Input Data and its shape,
                    Gabor filter parameters [F,phi,psi],
                    Gabor feature vector of values)
        reduction(+:ErrorCode)
{
   For each frequency F {
      ...
      Determine the resolution of the filter kernels
      Generate reusable convolution task object for
        convolution with the Gabor filter
      Generate convolution task object for
        convolution with the Gaussian filter
      ...

      #pragma omp for collapse(2)
      For each yaw angle psi {
        For each roll angle phi {
          ...
          Each thread writes the calculated feature value to
            the shared vector
          ...
        }
      }
   }
}
```

The above code distribute the 36 different angle parameter pairs to available threads which then individually calculate the associated Gabor feature value and writes to the Gabor feature vector that will be returned to the MATLAB program at the end of the function. The reduction clause checks that no error codes are returned.

After determining the resolution of the filter kernel, a reusable convolution task object can be used instead of the regular version as outlined in section 4.2.2 for additional performance.

Despite the reduction of task availability from 144 to 36, this is the taskfarm parallel strategy we use for the *fermi-0* and *phi* nodes which have core numbers between 12 to 16 due to the ability to reuse one of the convolution task objects.

4.6.4 OpenMP Parallelisation of All Parameters

To increase the concurrency suitable for many more threads, we can use the collapse directive on all three nested loops:

```
#pragma omp for collapse(3)
For each frequency F {
  For each yaw angle value psi {
    For each roll angle value phi {
      ...
```

so all 144 parameter combinations are available concurrently instead of just 36. This parallelisation scheme is better for when running more than 36 threads and this is attempted on the Intel Xeon Phi co-processor.

4.6.5 Use in MATLAB

This parallel strategy enables migrating much of the Gabor feature vector calculation from MATLAB to our multithreaded C-based routine .

The following MATLAB code shows that the features vector is now calculated by a MEX function that utilises the taskfarm parallel method utilising OpenMP and the MKL convolution routines.

```
function [ features ] = gabor_3d_features( subimg )

  %Gabor filter in 3d using 36 angles and 4 frequencies
  F = 25:25:100;
  B = [1.3,1,1,0.7];
  psi = (-60:30:90)/360*2*pi;
  phi = (-60:30:90)/360*2*pi;

  % Call MEX function
  features = gabor_3d_features_omp_mex(subimg,F,B,psi,phi,12);

  features = (features(:))';

end
```

so the MATLAB runtime calls a compiled MEX file called *my_imfilterCorr_omp*, passing the data, filter and the number of threads to run (12 in the example above). The variables `kc`, `ks` and `E` obtain answers from the parallel C-based routines instead. So we have replaced the original correlation routine with our own that utilises MKL to perform the correlation instead.

4.7 Chapter summary

In this chapter, we have stated the discrete convolution and correlation of two 3D matrices, one being the data and the other called a filter kernel. We covered the the full and same convolution and how Fourier transforms can be used to compute the convolution with less complexity with FFT implementations.

We have covered how to use the relevant Intel MKL functions to perform convolution and correlation calculations and how MATLAB can use our developed codes in C through the MEX framework.

Finally, we show two ways of parallelising the calculation, either by data decomposition or by taskfarming the different parameter combinations. The first method provides a replacement function for the original imaging code, whilst the second migrates much of the calculation of the Gabor feature vector from MATLAB to native C programs.

Chapter 5

Algorithm Benchmarks & Scalability

In this chapter we will look at the performance on our computational routines written in C that performs the correlation of two matrices using the Intel MKL libraries and using OpenMP to perform parallelisation by data decomposition. This examines the individual correlation calculations, as opposed to looking at the performance of the whole image application program.

The purpose of this is to test the scalability of one of our parallel strategies that divide the 3D data into smaller subcube domains and to assess its performance when applying the correlation directly and using FFTs. Both modes of correlation are offered by the Intel MKL library as described in the previous chapter. The work in this chapter also aims to provide an idea of expected performance when these native C routines are used with the original MATLAB program.

5.0.1 Summary of Methodology

The correlation of 3D medical image data together with a 3D filter kernel is in essence the correlation of 2 double-type 3D arrays. Hence for the benchmarking work in this chapter, we have developed and used a native C program that randomly generates an array for both the filter and the kernel so this would simulate the correlation of the real medical data and Gabor filter kernel. This then calls the main function that performs the correlation, passing in the necessary arguments that include those 2 randomly-generated arrays and the number of threads to use. Hence the C program essentially simulates what a compiled MEX file would do, which would call the same function that performs the correlation. The time taken to perform the calculation is determined by using the OpenMP function call that gets the wall clock time `omp_get_wtime()` and placed before and after the function call that performs the correlation.

Because of our use of OpenMP, we ensure that thread placement on all benchmarking runs are consistent by setting an environment variable `KMP AFFINITY` to ‘scatter’. This scheme round-robins the thread allocation so that consecutive threads alternate between the 2 sockets and amongst separate cores[1]. This ensures efficient resource utilisation possible amongst all running threads so the scheme ensures that, whenever possible, a thread has the resource of a whole CPU core.

To get a better idea of scalability, we run the C program on the *phi* compute-node with the Intel Xeons. This has 16 physical CPU cores available than the *fermi-0* compute-node which has only 12 physical cores. So we are able to examine the scalability upto 16 threads.

Throughout the benchmarking work done in this chapter, we have kept the size of the kernel filter the same whilst the image sizes change and the number of threads varies. We chose a 3D filter size of 14^3 points as this is one of the three filter kernel sizes in the real application.

5.1 Benchmarking Correlations in DIRECT mode

We will first at the performance of calculating the correlation formula directly on the 16 Intel Xeon CPU cores on the *phi* compute-node.

Figure 5.1 shows the variation in times to compute a correlation against the number of available threads for different data sizes[2]. Repeating the benchmark program provides close identical times so this enables us to quote a single benchmark run. Table 5.1 provides numerical times corresponding to Fig. 5.1 for a select number of threads.

	Input Data Size with 14^3 filter kernel (Time in seconds)			
# of Threads	$(64^2 \times 37)$	$(128^2 \times 75)$	$(256^2 \times 150)$	$(512^2 \times 300)$
1	0.359	2.845	24.244	198.316
2	0.177	1.399	11.257	92.117
4	0.081	0.684	5.690	46.382
8	0.045	0.383	3.153	25.673
11	0.071	0.401	2.484	20.251
13	0.085	0.464	2.716	17.618
16	0.025	0.204	1.665	13.512

Table 5.1: Time taken to correlate a data of specified size with the fixed size filter kernel for a select number of threads. The mode of operation of the MKL correlation function is set to compute it directly.

We see that for a single thread, the execution time grows exponentially as the input data size doubles. This can be seen by the approximate equal spacing on the log plot

[1]Thread Affinity Interface (Linux* and Windows*) - http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011Update/compiler_c/optaps/common/optaps_openmp_thread_affinity.htm, last accessed 21-08-2013.

[2]As a shorthand, the number in the labels in all the figures represent the data size by its first dimension. The actual data size can be found in the tables.

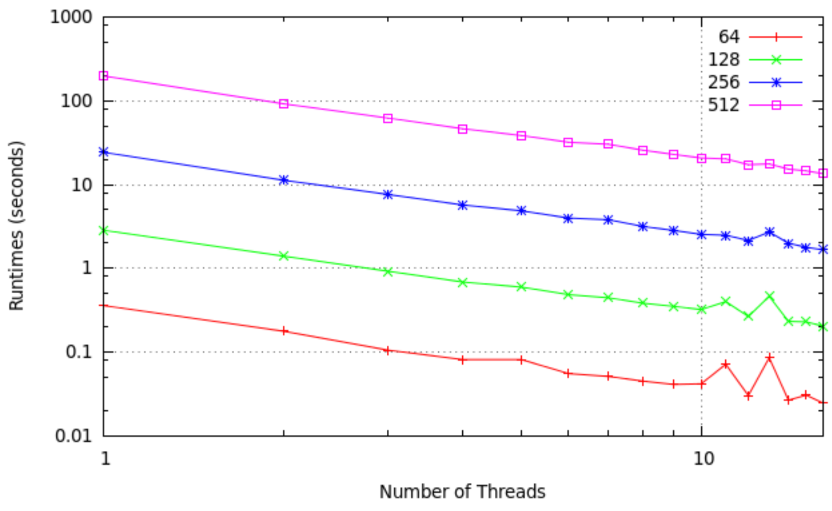


Figure 5.1: Graph showing the execution times of a single correlation with a 14^3 filter kernel against the number of running threads on the *phi* compute-node.

in Fig. 5.1 between each data size which doubles in each dimension. Figures 5.2 and 5.3 show the corresponding speed-up and parallel efficiency graphs respectively. Both these graphs conclude that the parallel strategy of dividing the input data into smaller subdomains amongst threads is very scalable on at least the Intel Xeon CPUs on the *phi* compute-node. However, what is noticeable is the drop in performance at thread counts 11 and 13. A closer investigation into this reveals a significant difference between the minimum and maximum subdomain size amongst threads in Table 5.2. This leads to one or more threads having to do a larger amount of work than the rest and thus time is wasted upon waiting for these threads to finish processing their chunk. All threads are synchronised at the end of the OpenMP parallel region, so the abnormal increase in runtime is because of waiting for those threads that are still working on their data. Hence

	Datasize $(64^2 \times 37)$		Datasize $(128^2 \times 75)$	
# of Threads	Min. Chunk	Max. Chunk	Min. Chunk	Max. Chunk
11	(5,64,37)	(14,64,37)	(11,128,75)	(18,128,75)
13	(4,64,37)	(16,64,37)	(9,128,75)	(20,128,75)

Table 5.2: The largest and smallest data chunk assigned to any thread when running the benchmark to produce Fig. 5.2.

this is the reason we see a sudden drop in performance. As the data size increases, the impact becomes less as more time is spent in computation. We have decided not to alter the decomposition algorithm to improve on this as we see the typical usage scenario where an even number of threads is chosen and ensures that an evenly balanced domain decomposition happens. To better illustrate the scaling performance, we present the speed-up and parallel efficiency graphs where we do not include the thread counts that produce imbalanced decompositions and the scaling picture appears better. Furthermore

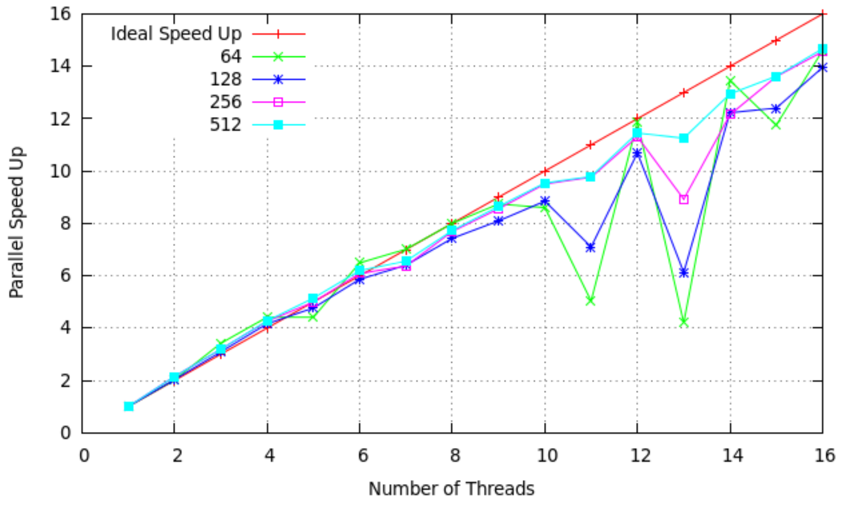


Figure 5.2: Speed-up obtained against the number of threads for Fig. 5.1.

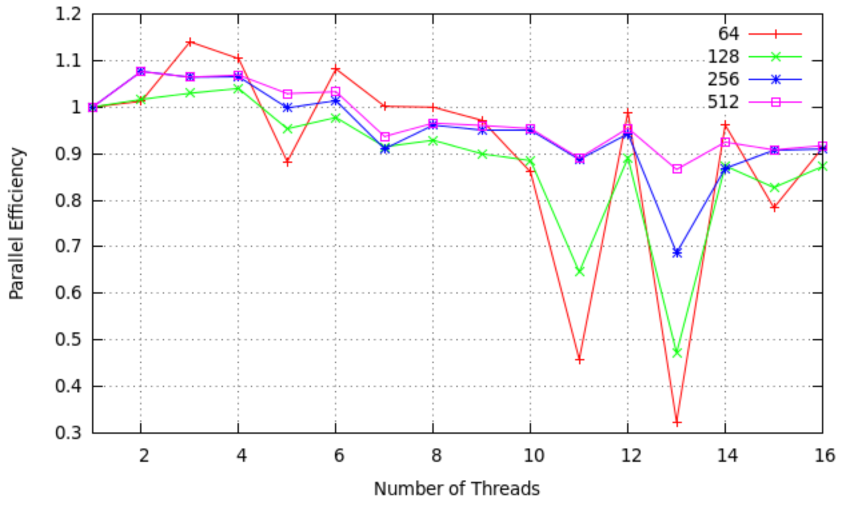


Figure 5.3: Graph showing the parallel efficiency obtained against the number of threads for Fig. 5.1.

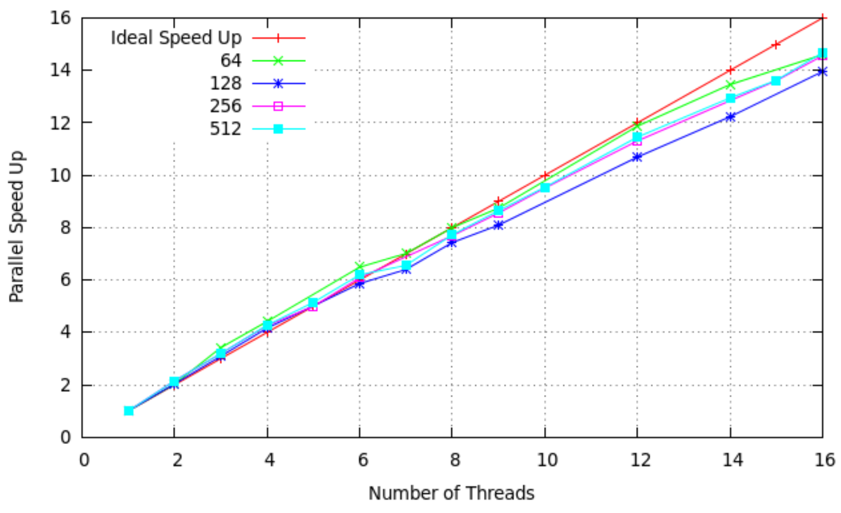


Figure 5.4: Graph showing speed-up against the number of threads for Fig. 5.1. Compared to Fig. 5.2, this omits the thread counts that result in imbalanced workloads amongst threads.

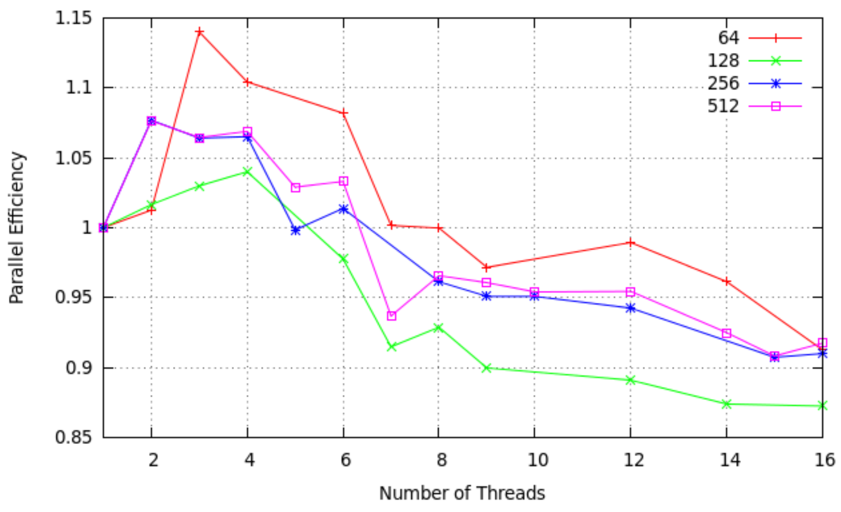


Figure 5.5: Graph showing the parallel efficiency against the number of threads for Fig. 5.1. Compared to Fig. 5.3, this omits the thread counts that result in imbalanced workloads amongst threads.

on these speed-up and efficiency graphs:

- For the first few threads, we see some super-linear speedup due to threads making use of additional on-chip cache as the second CPU on the node is employed and the possibility of using spare memory-bandwidth capacity that may not be fully used by one thread only.
- Near the end of the thread count, parallel efficiency deviates away from 1 due to some overheads of:
  - Initialising and setting up the OpenMP runtime environment.
  - Synchronisation overhead where the calculation does not complete until the last thread finishes. Even if the threads have a even workload, there will be the possibility of overheads such as OS noise or context-switching that causes a thread to take longer than expected.
  - Memory cache sharing amongst threads.

We conclude that our implementation of a parallel correlation function using Intel MKL routines together with OpenMP scales well relative to the performance of a single thread. This is as long as the thread count is such that the data division amongst them results in an evenly balanced workload.

5.2 Benchmarking Correlations in FFT mode

The alternative way to compute the correlation is to use Fast Fourier Transforms and it has been shown earlier that the convolution/correlation through the use of FFTs can dramatically reduce the runtime due to reduced computational complexity. In this section, we present the results of benchmarking the same function obtained by simply changing the mode of operation of the MKL function call to use FFTs instead.

Figure 5.6 shows the performance of the same parallel implementation as for the direct correlation calculations in the previous section, along with numerical runtimes in Table 5.3 for a select number of threads. This time we only show thread counts with a balanced workload.

By comparing the single thread times between Tables 5.1 and 5.3, the FFT method is faster than direct computation directly by a factor between 11 and 17, depending on the data size. This comes despite a drop in parallel scaling (Fig. 5.7) and efficiency (Fig. 5.8) due to how FFT algorithms perform for different data sizes. FFT algorithms favour certain data array lengths and we do not use any form of padding to achieve optimisation [21][26]. The nature of these parallel scaling and efficiency results are due to:

- The FFT scaling against data size, even for a single core, is not linear [27]. The peaks and troughs in Fig. 5.7 can be due to a thread taking longer to complete due to performance differences on different sized subdomains.

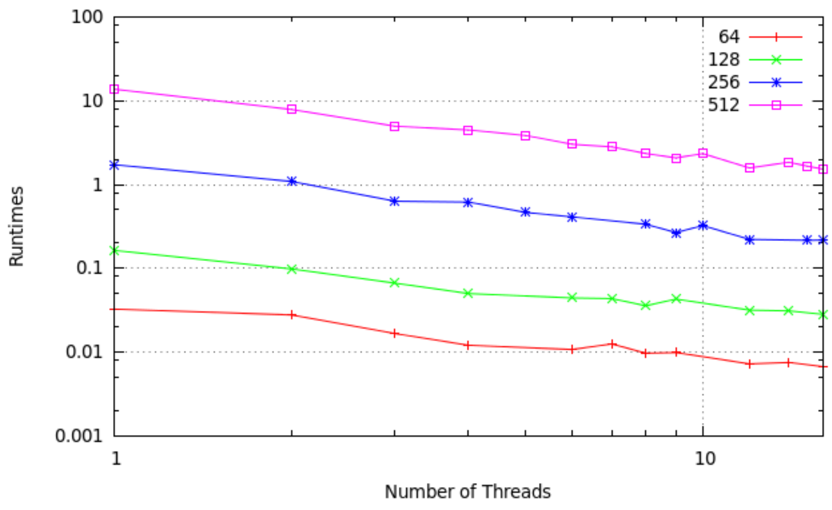


Figure 5.6: Graph showing the execution times of a single correlation with a 14^3 filter kernel against the number of running threads on the *phi* compute-node. This is the same benchmark run as Fig. 5.1 but the algorithm uses the Fast Fourier Transform method.

	Input Data Size with 14^3 filter kernel (Time in secs)			
# of Threads	$(64^2 \times 37)$	$(128^2 \times 75)$	$(256^2 \times 150)$	$(512^2 \times 300)$
1	0.032 (11.2)	0.164 (17.3)	1.726 (14.0)	13.760 (14.4)
2	0.028	0.098	1.088	7.885
4	0.012	0.050	0.617	4.485
8	0.010	0.036	0.336	2.348
16	0.007	0.028	0.217	1.531

Table 5.3: Time taken to correlate a data of specified size with a fixed size filter kernel for a selected number of threads. The mode of operation of the MKL correlation function is set to use FFTs. For a single thread, the speed-up over its counterpart in Table 5.1 is given in brackets.

- The lower parallel efficiency than the one for direct correlation indicates that FFT performance on decreasing data sizes becomes lower. For example, with 16 threads, the data division results in the smallest subcubes to each thread and more threads working on even smaller datasets do not result in a markedly increase in performance over 12 threads.

5.2.1 Correlation using multi-threaded MKL

Offered by the Intel MKL library is parallelisation of the correlation tasks using internal threading, achieved by compiling with the `-mkl=parallel` flag. This benefit is only offered when using FFT mode. This subsection shows the the performance obtained by benchmarking the serial version of our code compiled with the multi-threaded version

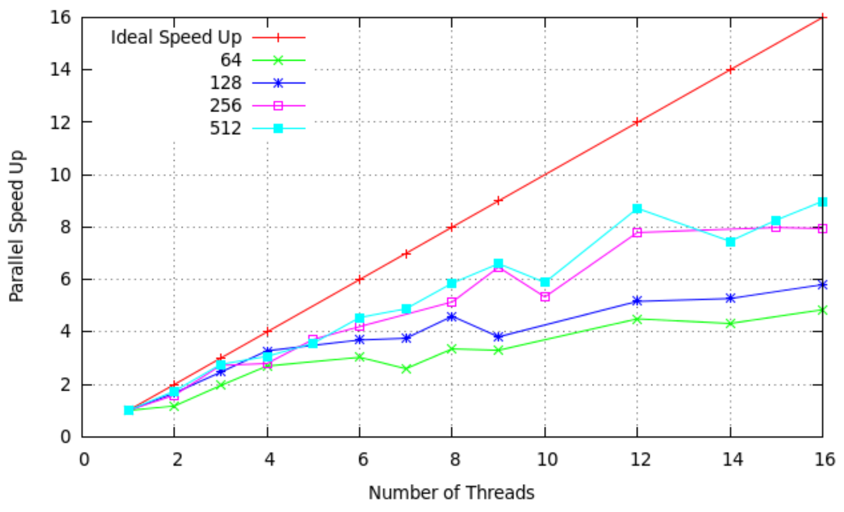


Figure 5.7: Speed-up against the number of threads for the correlation in FFT mode.

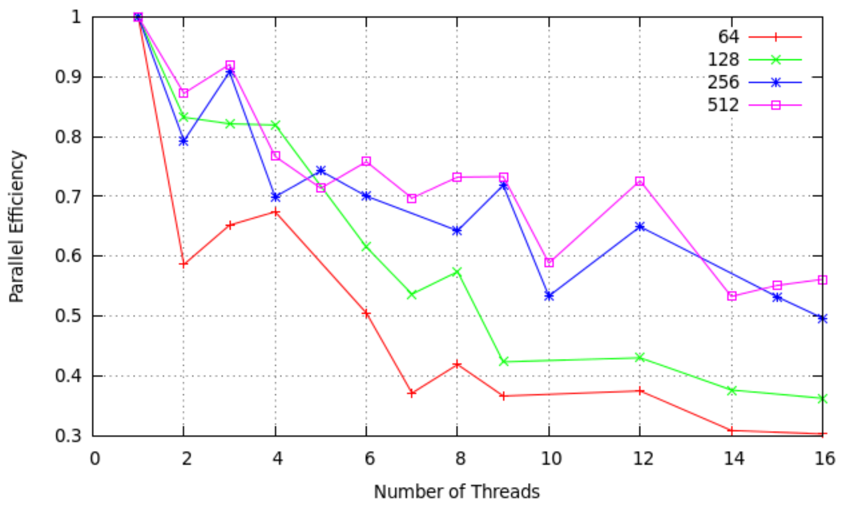


Figure 5.8: Parallel efficiency against the number of threads for the correlation in FFT mode.

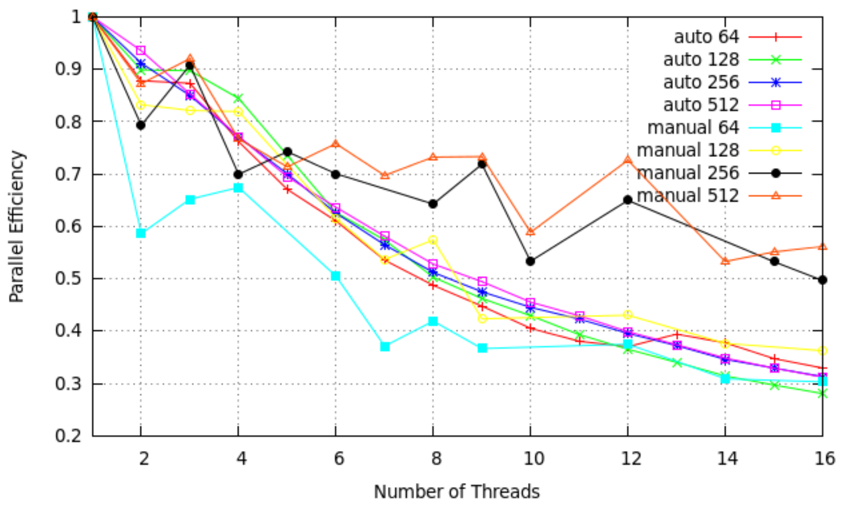


Figure 5.9: Graph showing the parallel efficiency of using multi-threaded MKL libraries and our parallel implementation.

of MKL.

Figure 5.9 compares the parallel efficiency with our manual data decomposition scheme for all 4 data sizes. We achieve a better efficiency metric for the largest data sizes, therefore making our parallelisation efforts worthwhile. However, this comes at the cost of a larger memory footprint due to each thread creating their own correlation tasks objects. For small data sizes, MKL offers better efficiency due to the overheads of our OpenMP data decomposition.

5.3 Error Analysis

This section shows the correctness of our OpenMP parallel implementation. For this analysis, we use our codes developed for GNU Octave due to its easier programming environment for fast development.

The code consists of Octave generating random 3D matrices for the data and filter and passing them to a compiled MEX file which in turn calls our parallel correlation code. The answer is compared to that generated by our serial version of the correlation code.

The relative error is calculated as

$$\max\left(\frac{|\mathbf{A}(i,j,k) - \mathbf{A_c}(i,j,k)|}{\mathbf{A_c}(i,j,k)}\right) \qquad \forall i,j,k \tag{5.1}$$

where $\mathbf{A_c}$ is the answer from our serial code.

Figure 5.10 shows the relative errors obtained for different numbers of threads and data sizes. The order is of 10^{-15} which is the limit of double precision and for 1 thread, the exact same answer as the serial code is produced.

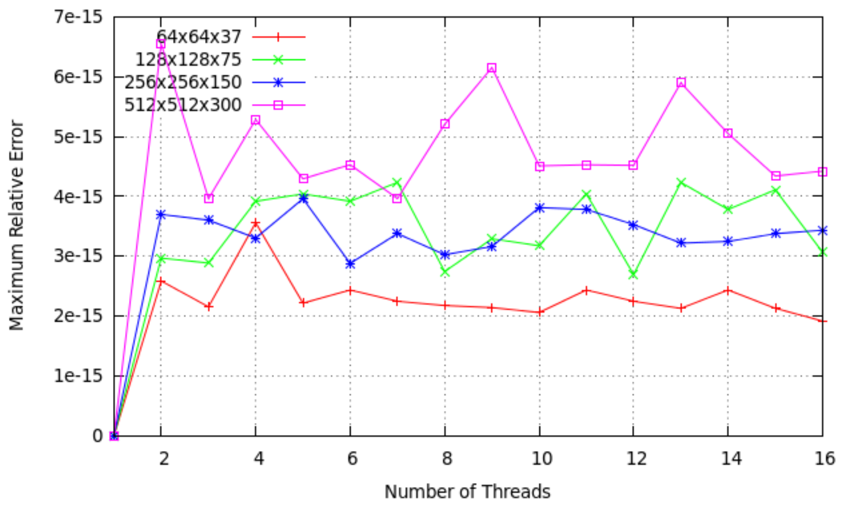


Figure 5.10: Graph showing the relative errors against the number of threads for different data sizes. The relative error is calculated from Eq. (5.1).

5.4 Chapter conclusions

The benchmarks have shown that computing the correlation using FFTs is many times faster than by direct computation. Hence the work in the next chapter just focuses on using the FFT mode for the convolution/correlation task objects involved.

When using FFT mode, we get reasonable scaling. The performance of using multi-threaded MKL that offers automatic parallelisation closely matches or outperforms our OpenMP scheme for the 2 smallest data sizes but we appear to achieve better performance for larger data sizes.

An error analysis concludes that our parallel code performs correctly on any number of threads and there appear to be no issues of thread-safety for different threads allocating their own memory space or executing correlation/convolution tasks in parallel.

As part of the next chapter, we replace the single function in the original MATLAB program responsible for correlation with the function benchmarked here and establish the overall performance gains as a result.

Chapter 6

Acceleration of original Imaging Application

This chapter presents the runtimes we are able to obtain or estimate by integrating our parallel C programs into the original MATLAB code. In the case of the compute-node containing the Intel Xeon Phi co-processors but without the MATLAB runtime, we estimate the theoretical gains possible by benchmarking the core computational routine using GNU Octave. This is unlike the previous chapter that only looked at the performance of convolution and correlations routines in isolation.

6.1 Performance of original MATLAB code on *fermi0*

We first look at the performance of the original MATLAB code from the hospital on the *fermi0* compute-node so we have a base time to build on for the rest of the chapter. This is done by timing how long it takes to calculate the Gabor feature vector of values only. Then this will be added on to the time it takes for MATLAB to perform the rest of the other texture feature calculations, giving us a total time for the whole original code that we intend to speed-up through parallelisation.

We used the method of inserting the MATLAB timer commands `tic` and `toc` around the Gabor filter function to avoid the added overhead of the in-built MATLAB profiler.

6.1.1 Gabor feature vector calculation

We time the execution of the Gabor feature calculation only for the original MATLAB code. All the three loops iterate over the three parameters of:

$$[4 \text{ frequencies } F,\ 6 \text{ yaw angles } \phi,\ 6 \text{ roll angles } \psi] \tag{6.1}$$

on the smallest dataset of (64x64x37) size are run through. But, as we have discovered in a preliminary study [8] that the larger data sizes can take hours or days to complete, we only run one angle combination pair for each frequency instead of all 36 and estimate the time taken by multiplying the time by 36. Each frequency is used in this estimate due to the different filter kernel sizes it generates, so the estimate is as accurate as possible by timing an angle pair for each frequency parameter. The timings are tabulated in Table 6.1.

Image Size	$(64^2 \times 37)$	$(128^2 \times 75)$	$(256^2 \times 150)$	$(512^2 \times 300)$
Runtime	774s	6090s (est.)	50249s (est.)	423150s (est.)

Table 6.1: Runtimes of processing the Gabor features vector by the original MATLAB code.

By timing the whole MATLAB program code but skipping over the Gabor feature calculation, the results are in Table 6.2.

Image Size	$(64^2 \times 37)$	$(128^2 \times 75)$	$(256^2 \times 150)$	$(512^2 \times 300)$
Runtime	5.06s	24.8s	198.1s	1552.6s

Table 6.2: Numerical runtimes of the original MATLAB code without the processing of the Gabor feature values.

6.1.2 Total Program Runtime

We establish the runtime of the entire MATLAB program by adding the Gabor feature calculation times in Table 6.1 to the time taken to calculate the other texture features.

Hence the total runtimes of the original MATLAB code on *fermi-0* for all four data sizes are summarised in Table 6.3 and illustrated in Fig. 6.1 by plotting the total runtime against the total number of pixels of the image.

Image size	Other Calc. (s)	Gabor Calc. (s)	Total Time (s)
$(64^2 \times 37)$ (151552 pixels)	5.06	774	779
$(128^2 \times 75)$ (1228800 pixels)	24.8	6090	6094
$(256^2 \times 150)$ (9830400 pixels)	198.1	50249	50447
$(512^2 \times 300)$ (78643200 pixels)	1552.6	423150	424700

Table 6.3: Runtimes of the original MATLAB program. Separate times are shown to calculate the Gabor feature vector and the other texture features to illustrate the Gabor feature calculations being the most expensive part of the program, which we target for parallelisation.

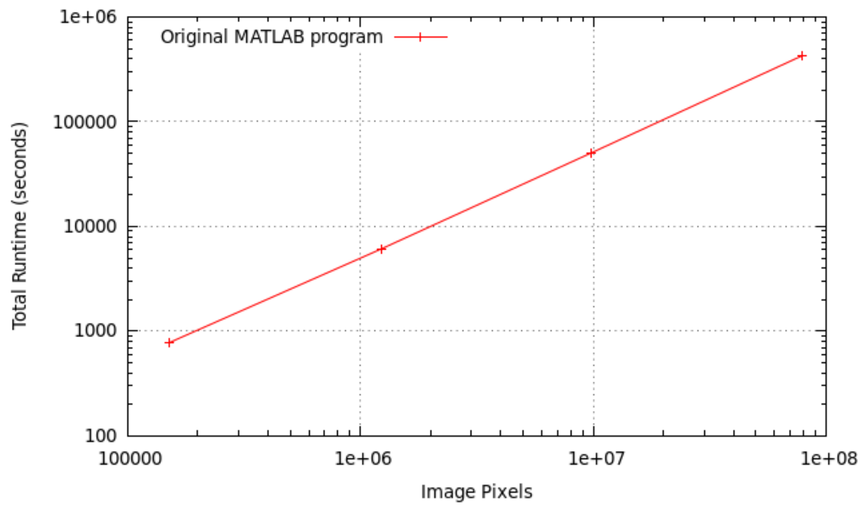


Figure 6.1: Log plot of the total runtime of the original MATLAB image processing program against the number of pixels of the input image data.

These figures confirm that the calculation of the Gabor feature values, involving the correlation, of the images is the most expensive part which we have targeted for parallelisation in the next sections of this chapter. We also conclude that MATLAB's filtering routine computes the correlation directly, rather than through use of FFTs which will become clear later on in this chapter.

6.2 Accelerating the MATLAB program

This section will show the reduction in runtimes we are able to achieve using the original MATLAB program. By replacing the computationally expensive routines within the MATLAB code with functions that call native C compiled code, we provide the benefit of using the Intel MKL routines instead as well as performance through parallelisation. The next two subsections will look at the runtimes obtained through two different ways:

- Replacing the single MATLAB function responsible for correlation between two matrices with an equivalent function utilising the Intel MKL with multicore parallelisation.
- Replacing most of the Gabor feature calculation code with a equivalent code in C that computes and returns the feature vector to the original MATLAB program.

6.2.1 Results by Function Substitution in MATLAB

This looks at the gains in performance by simply replacing the MATLAB routine `imfilter` with a MEX function that equivalently computes the correlation using the Intel VSLCorr routines in C. This method of function substitution offers a simple way to accelerate the existing MATLAB code as the user simply changes any instances of `imfilter` whilst continuing to use MATLAB programming for other code or logic as desired.

Using this acceleration strategy, Table 6.4 shows two runtimes obtained by running the calculation twice separately. Generally, there is a reduction in runtime with additional threads, except for the smallest dataset. Immediately, the times compared to the original are an order of magnitude lower due to the Intel MKL correlation routines using FFTs to perform the filtering on the image data which is of less complexity than direct calculation of the correlation formula.

	Input Data Size (Times in seconds)			
# of Threads	$(64^2 \times 37)$	$(128^2 \times 75)$	$(256^2 \times 150)$	$(512^2 \times 300)$
1	13.19, 13.13	105.64, 104.68	805.81 (802.40 est.)	(7972.67, 7970.83 est.)
2	10.10, 10.10	66.74, 65.84	530.53, 529.90	(4433.44, 4429.48 est.)
3	10.69, 10.33	60.49, 59.79	371.16, 370.40	(2997.45, 2995.01 est.)
4	8.27, 8.20	43.63, 41.45	331.44, 329.34	(2488.39, 2487.62 est.)
6	8.58, 8.29	41.22, 39.74	250.32, 249.61	(1765.95, 1764.84 est.)
9	10.78, 10.22	50.99, 47.44	243.09, 237.61	(1412.81, 1404.18 est.)
12	11.04, 9.54	32.30, 30.26	196.50, 194.07	1296.26 (1295.83 est.)

Table 6.4: Runtimes of processing the Gabor features vector by function substitution of the MATLAB routine with a MEX function that calls equivalent Intel MKL routines. For long runtimes, the estimation method we used for the original code in section 6.1 applies here.

Two other interesting metrics are the parallel speed-up and efficiency of the results in Table 6.4 which are shown in Figs. 6.2 and 6.3 respectively. From these graphs, we comment that for small data sets it is not worth parallelising due to the correlation in FFT mode can complete on the order of milliseconds and so we see poor performance on many threads. Generally, as the data size becomes larger, efficient use of cores improves. We believe the levelling off of performance is due to the internal FFT algorithm preferring larger datasets to work on and here the subdomains become smaller with increasing number of threads.

Table 6.5 shows the estimate for the whole MATLAB program for a selected number of threads by adding on the time for the non-Gabor related calculations in Table 6.2 to the times in Table 6.4, using the longest time of each pair. The number in brackets is the speed-up over the total runtime of the original MATLAB program in Table 6.3 and Figs. 6.4 and 6.5 illustrates the share of the Gabor feature calculations with the runtime of the other texture calculations when using 12 threads. This shows that the reduction in runtime through the change in the algorithm to use FFTs in addition to parallelisation becomes the less dominant calculation component for larger images.

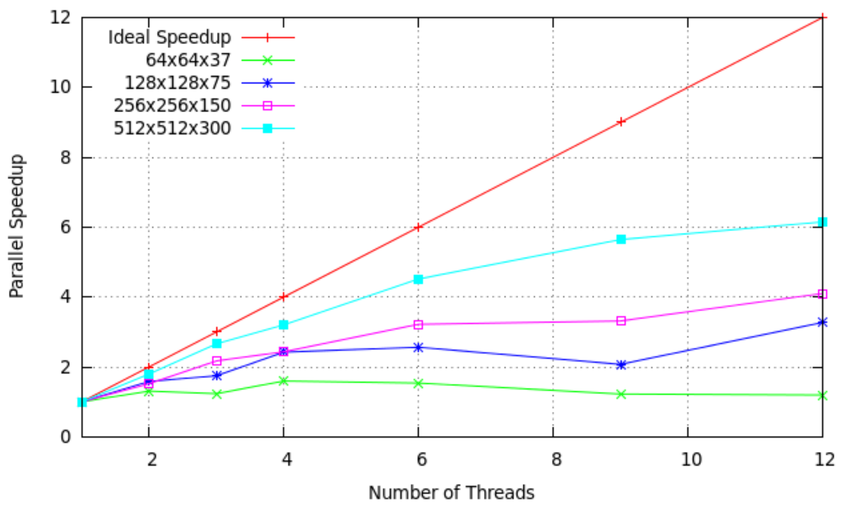


Figure 6.2: Graph showing the speed-up metric of this parallel strategy for the 4 different image sizes. The longest time obtained in Table 6.4 is used.

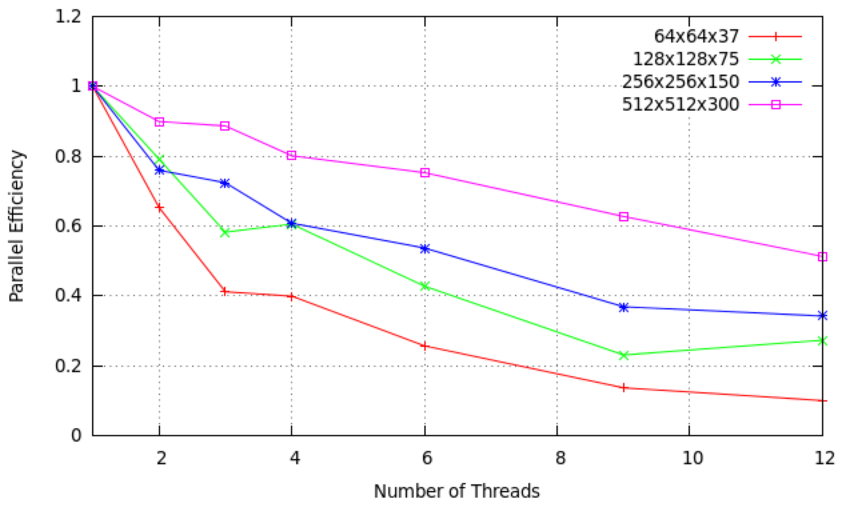


Figure 6.3: Graph showing the parallel efficiency metric of Fig. 6.2 for the 4 different image sizes.

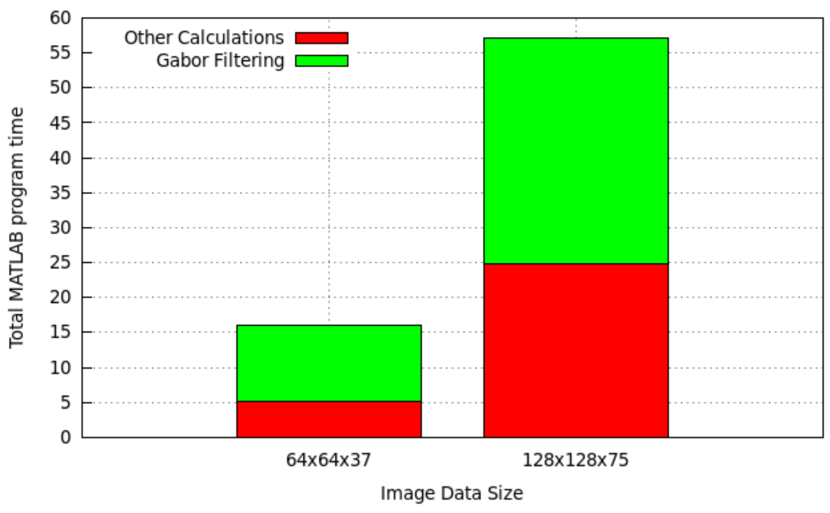


Figure 6.4: Barchart showing the runtime share between the Gabor feature calculations and the rest of the other texture calculations for the two smallest image datasets. The times are for the case using 12 threads.

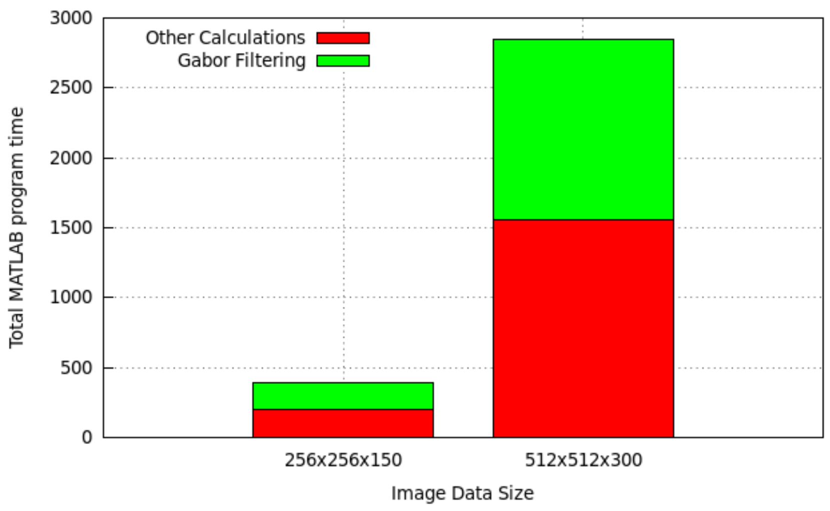


Figure 6.5: Barchart showing the runtime share between the Gabor feature calculations and the rest of the other texture calculations for the two largest image datasets. The times are for the case using 12 threads.

	Input Data Size (Time in seconds)			
# of Threads	$(64^2 \times 37)$	$(128^2 \times 75)$	$(256^2 \times 150)$	$(512^2 \times 300)$
1	18.25 (42.7)	130.44 (46.7)	1003.91 (50.3)	9525.27 (44.6)
4	13.33 (58.4)	68.43 (89.1)	529.54 (95.3)	4040.99 (105.1)
12	16.10 (48.4)	57.10 (106.7)	394.6 (127.8)	2848.86 (149.1)

Table 6.5: Runtimes of the entire MATLAB program by adding the non-Gabor runtimes with the reduced runtimes in Table 6.4 for 1, 4 and 12 threads (using the time obtained on the left). The total runtime of the original program in Table 6.3 is reduced by a factor of the number in brackets.

Multi-threaded MKL Performance

By compiling the serial version of the code for the replacement function with the parallel MKL flag, we can test the performance of the automatic parallelisation capability offered by the library. When the single thread creates and then executes the correlation task objects, the MKL library employs multi-threading internally.

Again, the MATLAB `imfilter` routine is substituted with a MEX function that calls our serial version of the correlation code, where the underlying code that creates and executes MKL correlation task objects has no OpenMP directives.

Table 6.6 shows the times obtained when using the multi-threaded version of the MKL library. We try to make use of all 24 threads available on the *fermi-0* compute-node by calling the MKL function *mkl_set_num_threads(24)*. The results show that for very small image data, enabling automatic parallelisation is preferable to manual parallelisation by data decomposition. This is probably due to the small calculation time for each thread and thus the overhead of 12 threads synchronising their results at the end is larger than MKL's internal threading routine. For larger datasets, our parallel scheme matches or outperforms the automatic parallelisation where data distribution of smaller (but not too small) subcube data to threads seems to exhibit better performance.

	Input Data Size (Times in seconds)			
# of Threads	$(64^2 \times 37)$	$(128^2 \times 75)$	$(256^2 \times 150)$	$(512^2 \times 300)$
24	5.86, 5.49	32.75, 32.61	213.60, 213.48	1674.4
Ratio of the time for 12 threads in Table 6.4 to the time here	1.88, 1.74	0.99, 0.93	0.92, 0.91	0.77

Table 6.6: Runtime of Gabor feature calculation routine that is compiled with the multi-threaded version of the Intel MKL library. Times are obtained by running twice except for the largest dataset. The ratio indicates whether the performance of multi-threaded MKL is faster or slower than our parallel method.

6.2.2 Numerical Error Analysis

Having achieved a large reduction in total runtime over the original program by a factor of upto 149, checking the numerical accuracy of our parallel solution is also required. In this subsection, the Gabor feature values are compared against the original MATLAB data for the 64x64x37 data size sample.

Figures 6.6 and 6.7 show the absolute and relative error differences between the output of our parallel solution and the original MATLAB program. The absolute error is of the order $\mathcal{O}(10^{21})$, however the relative error is of order $\mathcal{O}(10^{-14})$, which is close to the limit of double-type numerical precision on computers (16 significant figures). For further illustration, the maximum and minimum values for both our solution and the original code is in Table 6.7 using the 'long e' format option within MATLAB. In this

Function	Minimum Feature Value	Maximum Feature Value
MATLAB	$9.369675706004886 \times 10^{29}$	$7.329071119659973 \times 10^{36}$
Intel MKL	$9.369675706004892 \times 10^{29}$	$7.329071119659978 \times 10^{36}$

Table 6.7: The maximum and minimum Gabor feature values generated by the original MATLAB program and our parallel solution. The full double-precision number format is shown for comparison.

table, the last one or two decimal places are different hence the uncertainty goes from beyond 16 to 14 significant figures when comparing answers with the original output of the hospital imaging program. It remains to be seen if this leads to a different qualitative outcome in terms of ultimately predicting post-treatment side effects for lung cancer patients. But a discussion with the code providers indicated that this would not have an effect. So we do not expect the performance improvements to compromise the results generated by the code.

6.2.3 Results by Taskfarm Parallelism

This subsection presents the runtimes obtained using the second of the two parallel strategies. This strategy involves replacing much of the original Gabor feature calculation MATLAB code with a MEX function that will take the frequency and angle parameters in expression (6.1) and the input data. Hence the main body consisting of 3 loops in the original code is migrated to a native C programs and returns the 144-length vector of Gabor feature values to the MATLAB environment. Hence this strategy is called the taskfarm parallel strategy as this mainly involves many CPU cores calculating a Gabor feature value for a particular parameter combination. For the rest of this chapter, we will simply refer to this as the **taskfarm** method as a shorthand way.

The results presented here are for the code that distributes the 36 different angle pairs to individual CPU cores. Due to only 12 cores available on *fermi-0*, this divides evenly into the 36 tasks. The code is able to take advantage of correlation task object reuse

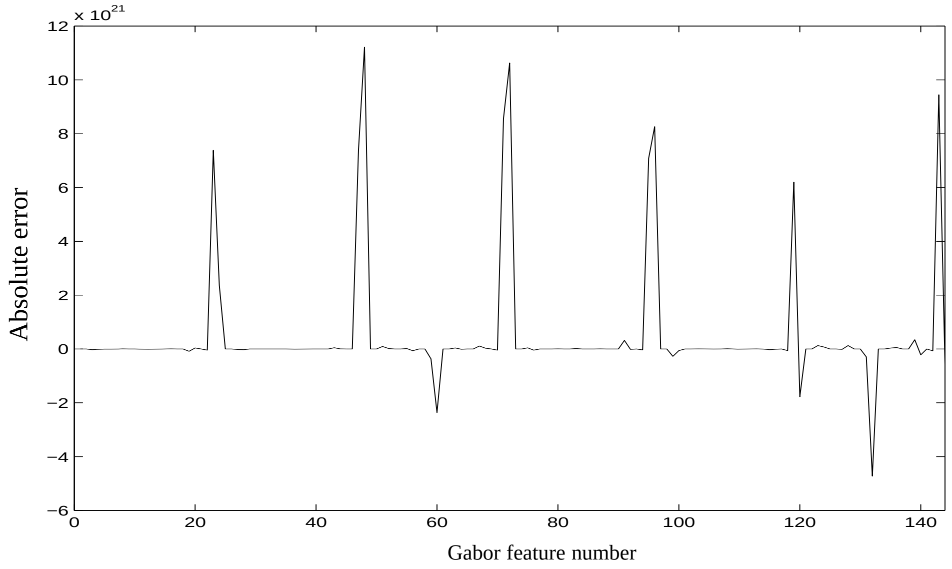


Figure 6.6: Plot of the absolute error between the 144-length Gabor feature vector generated by our parallel solution in section 6.2.1 (using 12 threads) and the original MATLAB program for the 64x64x37 sample image data.

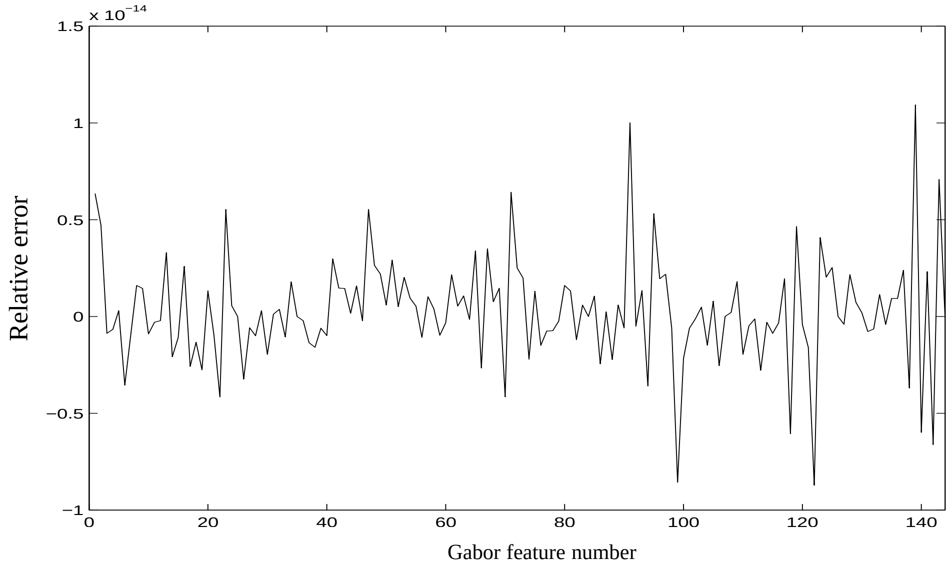


Figure 6.7: Plot of the relative error between the 144-length Gabor feature vector generated by our parallel solution in section 6.2.1 (using 12 threads) and the original MATLAB program for the 64x64x37 sample image data.

as shown in section 4.6.3 where the task correlates for different filters with different angles before the task object is destroyed and a new one created for the next frequency parameter. The choice of thread numbers are based on the fact that they divide nicely into the 36 available tasks related to the 2 sets of 6 angles in expression (6.1) so that maximum thread concurrency is maintained.

Table 6.8 shows two runtimes to calculate just the Gabor feature vector using this parallel method by running the calculation twice, except for the largest dataset where the runtimes are long. After twelve threads, the serial code without OpenMP compiler directives is compiled with the multi-threaded MKL library and executed with all 24 available threads, 12 of which are virtual threads in simultaneous multi-threading (SMT). The times obtained are shown on the last row of Table 6.8.

Here the general trend is a reduction in the time-to-solution as more cores become available to process Gabor feature values at the same time. Unfortunately, this method of each thread creating separate correlation tasks for themselves has a large memory footprint, such that the 48GB of system memory on the *fermi-0* compute node is insufficient for 12 cores. Using the serial code compiled with the multi-threaded MKL library provided a workaround, allowing to process the largest image, however performance is less than that obtained for 9 threads using our parallel method.

	Input Data Size (Times in seconds)			
# of Threads	$(64^2 \times 37)$	$(128^2 \times 75)$	$(256^2 \times 150)$	$(512^2 \times 300)$
1	12.34, 12.20	104.06, 101.86	772.53, 772.48	(7795.44, 7787.05 est.)
2	7.35, 6.99	53.07, 52.90	398.72, 396.72	4011.62
3	4.92, 4.90	36.26, 36.08	273.91, 272.70	2716.42
4	4.04, 4.01	28.18, 28.18	209.68, 209.49	2087.46
6	3.06, 3.05	17.79, 19.76	149.88, 149.37	1456.17
9	2.71, 2.81	20.00, 20.09	147.36, 144.23	1044.88
12	2.05, 2.03	13.88, 13.71	102.61, 102.06	Not enough memory
24*	6.06, 5.69	33.88, 33.25	223.41, 223.39	1755

Table 6.8: Runtimes of processing the Gabor features vector by distributing the different angle parameters to different threads and returning the Gabor feature value associated with that particular parameter combination. *Serial code is used with MKL performing internal multi-threading with 24 threads.

Figures 6.8 and 6.9 give the parallel speed-up and efficiency for this taskfarm parallel strategy, using the longer times in Table 6.8. Compared to the metrics of the first parallel strategy in Figs. 6.2 and 6.3, this taskfarm strategy exhibits better performance metrics for all four data sizes. This strategy suits smaller datasets so that each thread works on the whole data for a particular generated Gabor filter, rather than further dividing the small dataset into even smaller pieces of data amongst the threads.

By comparing Table 6.8 with the Gabor feature calculation times obtained in Table 6.4 where only a single function substitution is performed, the following differences are observed:

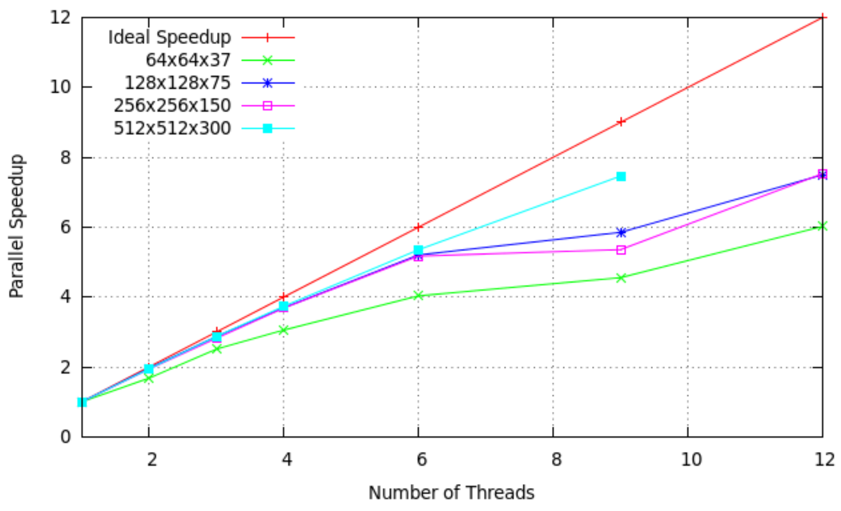


Figure 6.8: Graph showing the speed-up metric of this parallel strategy for the 4 different image sizes. Table 6.8 is used with the left-hand most times among the ranges.

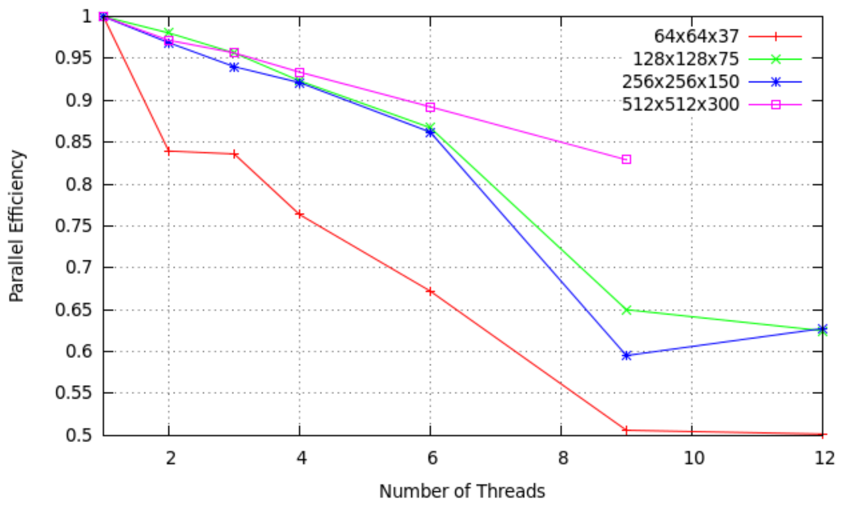


Figure 6.9: Graph showing the parallel efficiency metric of Fig. 6.8 for the 4 different image sizes.

- For single thread performance, the taskfarm method has higher performance as expected due to the bulk of the work now done with a compiled C program. This includes the three nested loops with the main inner-loop body containing the generation of the Gabor filter kernel and intermediate calculations. These calculations perform faster as expected.
- For the smallest dataset, 64x64x37, the taskfarm method continues to improve beyond 4 threads. This is due to the lower overhead of getting separate cores to perform correlation on the whole data, which already takes milliseconds to do on a dataset this size. This is in addition to improvements where the logic and computation is performed in a compiled C program rather than within MATLAB.
- Comparing the performance with 12 threads, the taskfarm method achieves the faster time. However, this is at the expense of using a larger memory footprint to calculate the Gabor features in parallel. In addition, these task objects return answers in separate memory arrays, so the MKL VSLCorr library does not produce an in-place solution where the answer replaces the original input data. As the correlation produces a complex-array, twice the storage is required.
- Despite being unable to run this scheme with all 12 available cores, the runtime obtained for nine threads is faster than using 12 threads with the function substitution method. This concludes that, at the cost of higher memory use, the taskfarm parallelisation method exhibits higher parallel efficiency. This is due to the possibility that, in the first parallel method where data decomposition into subdomains for different threads occur, the internal FFT routine may perform slower. This is due to the particular block size of the subcube of data that each thread get. FFT algorithms prefer data blocks of a certain size and optimisation tips include padding the data to a certain size. In addition, FFTs tend to perform slower on smaller datasizes whereas they prefer working on larger datablocks [27].
- Finally, compiling and linking the C codes with the multithreaded MKL library provides a workaround for the memory limitation as all 12 cores running 2 threads were able to complete the calculation successfully. However, its performance is equivalent to our taskfarm method using between 3 and 4 threads.

Finally, we summarise the best case performance obtained using this parallelisation method by showing the total MATLAB program runtime. Table 6.9 shows the share in runtimes between Gabor feature calculation and the other calculations. This shows the results using 12 threads except for the largest data size where only 9 threads will run. Figures 6.10 and 6.11 provides an illustration of the share of the total runtime.

Compared to using the first parallel strategy, all image size cases have reduced runtimes to the point that they account for less than half of the total MATLAB runtime, leaving potential future optimisations to target other calculation routines such as Haralick features. In the first strategy, only the larger two data sets achieved this, seen in Fig. 6.5.

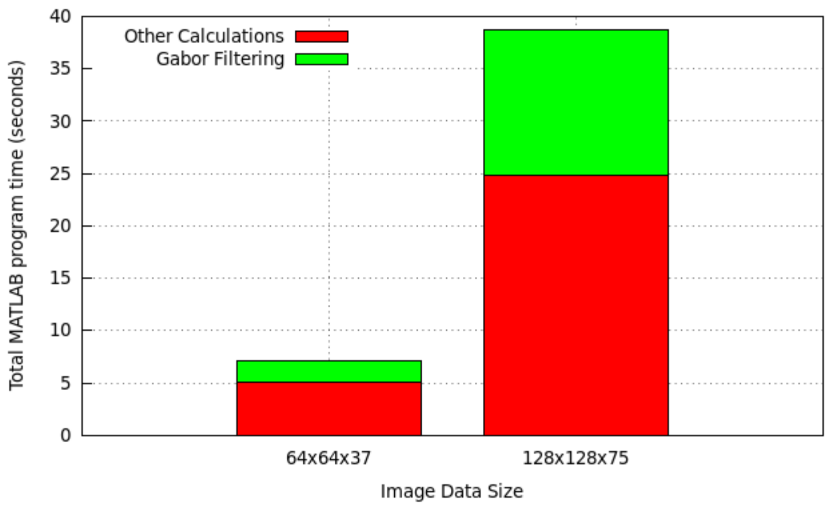


Figure 6.10: Barchart showing the runtime share between the Gabor feature calculations and the rest of the other texture calculations for the two smallest image datasets in Table 6.9.

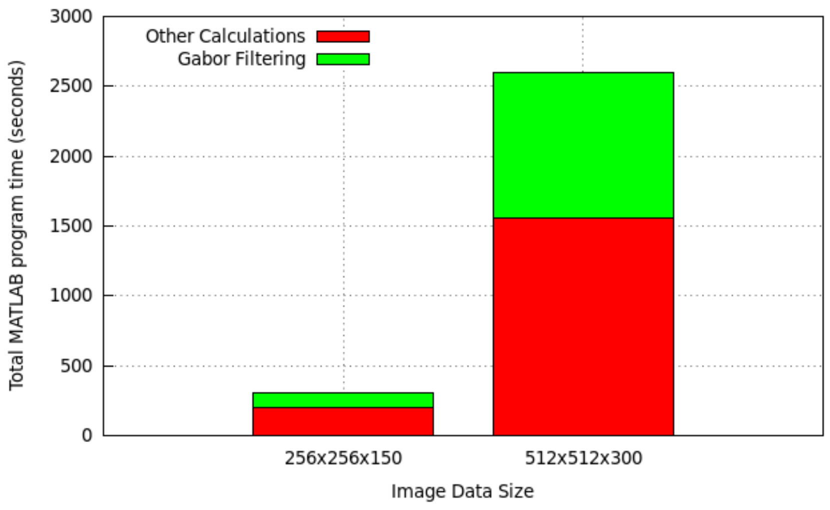


Figure 6.11: Barchart showing the runtime share between the Gabor feature calculations and the rest of the other texture calculations for the two largest image datasets in Table 6.9.

Image size	Other Calc.	Gabor Filtering	Total Time
64x64x37	5.06s	2.05s	7.11s (109.6)
128x128x75	24.8s	13.88s	38.68s (157.5)
256x256x150	198.1s	102.61s	300.71s (167.8)
512x512x300	1552.6s	1044.88s (on 9 threads)	2597.48 (163.5)

Table 6.9: Table showing the share of the runtime in seconds between the Gabor and non-Gabor related feature calculations. The number in brackets is the reduction factor achieved on the total runtime of the original MATLAB program in Table 6.3.

6.2.4 Numerical Error Analysis

Following the same methodology that we did for the first parallel strategy, we check the accuracy of our answer for the 144-length Gabor feature vector against the original MATLAB data for the 64 datasize sample. The number of threads we used is 12 as we did in the analysis for the first parallel strategy. The absolute and relative errors of the 144-feature vector are shown in Figs. 6.12 and 6.13 respectively.

The same conclusions as for the first parallel strategy applies in this case but the numerical difference here is a further two decimal places. We can see that the absolute error is of the order $\mathcal{O}(10^{23})$ and the relative error order is $\mathcal{O}(10^{-12})$. Table 6.10 shows the maximum and minimum value of the calculated Gabor feature vector for both this parallel method and the original MATLAB program.

Function	Minimum Feature Value	Maximum Feature Value
MATLAB	$9.369675706004886 \times 10^{29}$	$7.329071119659973 \times 10^{36}$
Intel MKL	$9.369675706000266 \times 10^{29}$	$7.329071119659139 \times 10^{36}$

Table 6.10: The maximum and minimum Gabor feature values (in full double-precision format) generated by the original MATLAB program and our taskfarm parallel solution.

This table shows that the numerical difference between the answers is now two more decimal places than in the first analysis in Table 6.7, so the margin of uncertainty compared to the original MATLAB program is now from 16 to 12 significant figures. Again, it remains to be seen if this affects the qualitative outcome of the whole medical objective in predicting the occurrence of pneumonitis for patients.

6.2.5 Section summary and conclusions

We summarise the work done with the original MATLAB program with the the enhancements we have done to accelerate and parallelise the most computational part of the program:

- Two parallel schemes were used to achieve speed-up. One provided the convenience of replacing the single MATLAB routine with an alternative that called

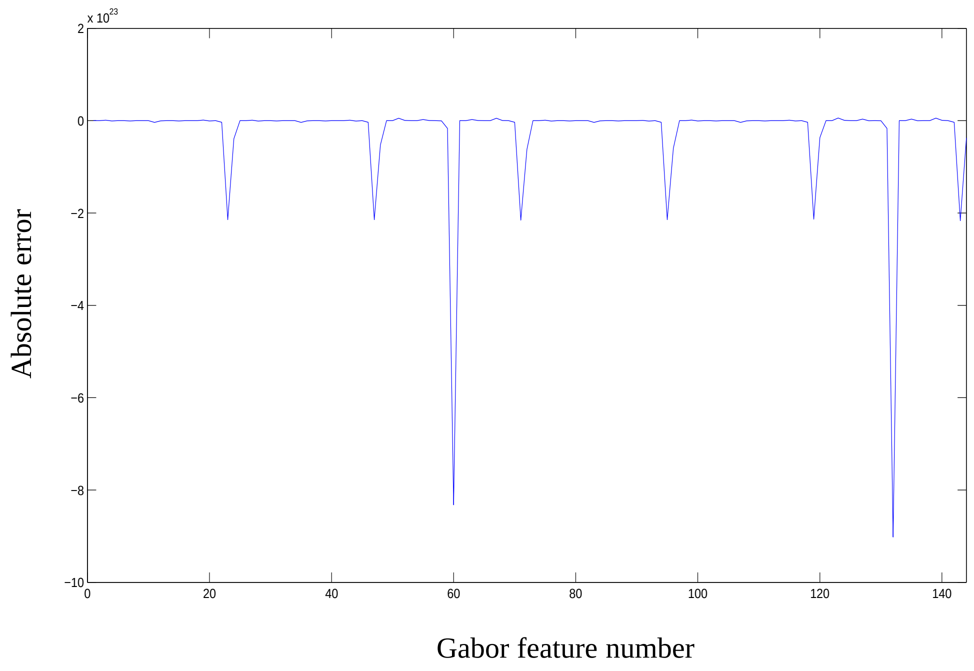


Figure 6.12: Plot of the absolute error between the 144-length Gabor feature vector generated by our parallel solution in section 6.2.3 (using 12 threads) and the original MATLAB program for the 64x64x37 sample image data.

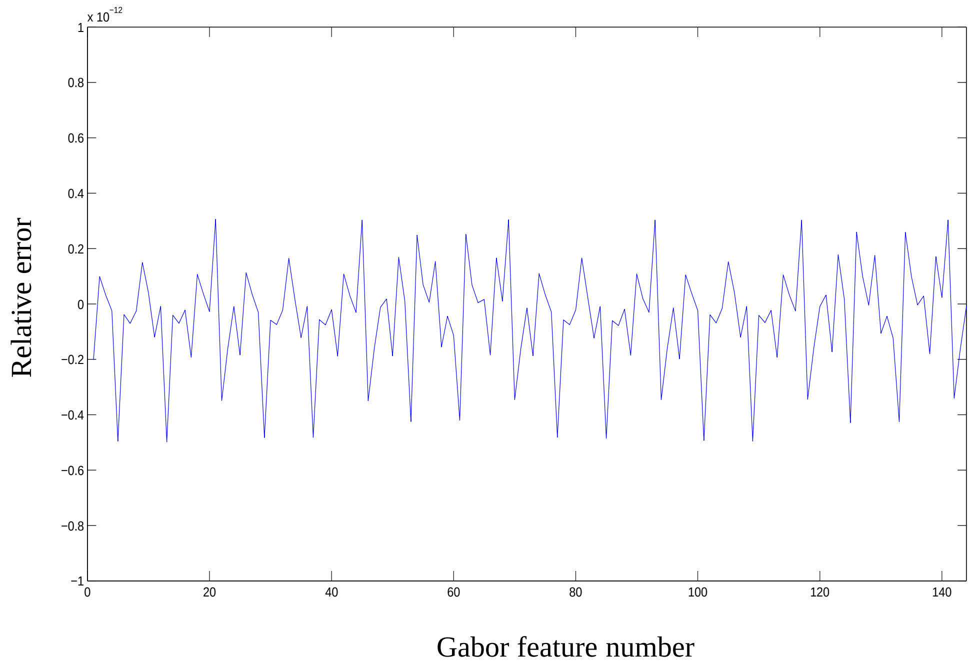


Figure 6.13: Plot of the relative error between the 144-length Gabor feature vector generated by our parallel solution in section 6.2.3 (using 12 threads) and the original MATLAB program for the 64x64x37 sample image data.

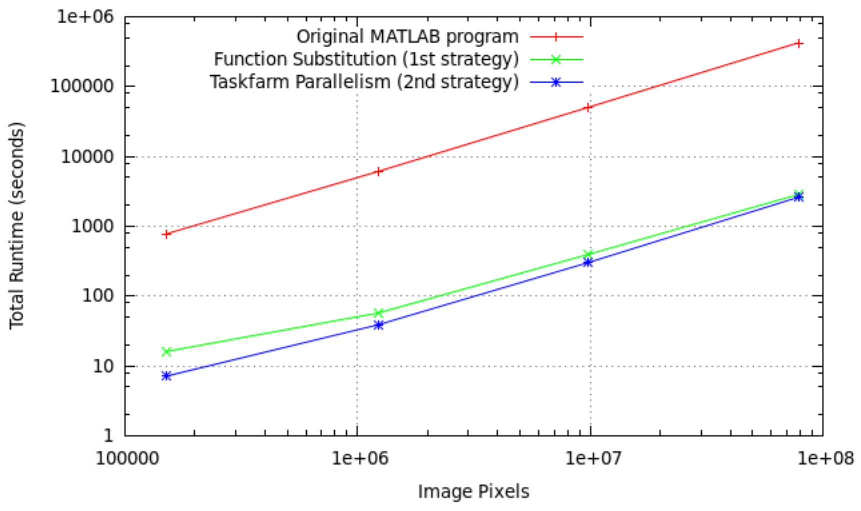


Figure 6.14: Log plot of the total runtime of the entire MATLAB program. This compares the original code with the enhancements using the MKL libraries under the two different strategies discussed.

equivalent routines in the Intel MKL library, the other was a taskfarm parallelisation strategy where more of the computation was moved to the external C program.

- We have successfully reduced the runtime of the original MATLAB program and achieved speed-up of between 42.7 and 167.8 depending on the data size worked on and the number of threads used. Figure 6.14 shows the times of the original MATLAB program against the number of image pixels together with the total runtimes for the two parallel strategies. This concludes that the original MATLAB code was performing the 3D correlation directly which is of a higher complexity than FFTs and hence leads to a poorer performance.

- With the very large speed-ups achieved, the numerical errors between the original answer and answers produced from both parallel schemes show that they work correctly due to the small relative error. The answer differs beyond 13 significant figures in the worst case and 15 in the best case.

- Both parallel schemes have good parallel speed-up and efficiency metrics, except when using the smallest data size with the first parallel scheme. In that case, the multi-threaded capability of the Intel MKL library provides better runtimes but for other datasizes, our own parallel schemes provide better performance.

- The taskfarm parallel scheme provides a lower runtime than the other one for the largest data size at the cost of a large memory footprint, where only nine threads can run successfully.

6.3 Performance Metrics using CPUs on the *phi* node

In this section, the best case results for both parallel schemes will be summarised. In addition, we will show the lack of concurrency of the taskfarm scheme and demonstrate running an alternative that distributes all 144 parameter combinations in parallel. A profile of the taskfarm parallel is briefly mentioned.

6.3.1 Results of both parallel schemes

Due to MATLAB unavailable on this compute-node, the runtime results were obtained by running the Gabor feature calculation part using GNU Octave.

	Input Data Size (Time in seconds)			
# of Threads	$(64^2 \times 37)$	$(128^2 \times 75)$	$(256^2 \times 150)$	$(512^2 \times 300)$
1	10.022 (9.66)	83.792 (82.5)	669.778 (671)	6858.83 (6815)
2	7.968 (7.24)	52.316 (46.4)	424.523 (423)	3884.45 (3889)
4	6.744 (6.10)	34.236 (29.2)	291.163 (287)	2318.53 (2312)
8	7.295 (6.61)	27.385 (23.0)	216.393 (212)	1451.52 (1450)
16	7.666 (7.07)	28.583 (22.0)	162.690 (158)	1103.47 (1100)
16*	4.47s	26.98s	217.13s	1761.21s

Table 6.11: Time taken to calculate all 144 Gabor feature values by function substitution in Octave on the *phi* node using the CPUs, obtained by individually timing the code that computes the Gabor feature vector. For the two largest data sizes, the times in brackets are estimated from computing 4 Gabor feature values. For the smallest 2 sizes, the bracket times are obtained as part of a benchmarking code. *Code compiled with multi-threaded MKL library.

Table 6.11 shows the runtimes of the first parallelisation strategy obtained in the following ways. All the runtimes quoted without brackets are obtained by running code that sets the number of threads to use and then times the calculation of the Gabor feature vector. Hence a program is run with the following pseudo-code inside:

```
set number of threads N
   ...
t1 = start-time
for each Gabor filter parameter (F,psi,phi)
{
   ...
   calculate-correlation(data, filter(F,psi,phi), number of threads N)
   ...
   Gabor feature(F,psi,phi) = final result
   ...
}
t2 = end-time
time-taken = t2-t1
```

```
print time to screen
end program
```

For the smallest 2 datasets 64x64x37 and 128x128x75, the runtimes in brackets are obtained as part of a benchmarking code where the Gabor feature vector is calculated within a loop body that increments the number of threads to use from 1 to 16:

```
for each N from 1 to 16  // Loop over the number of
{                        // threads sequentially
    ...
  t1 = start-time
  for each Gabor filter parameter (F,psi,phi)
  {
      ...
    calculate-correlation( data, filter(F,psi,phi),
                           number of threads N)
      ...
    Gabor feature(F,psi,phi) = final result
      ...
  }
  t2 = end-time
  time-taken = t2-t1
  print time to screen
    ...
}
end program
```

This was to investigate any difference between obtaining times using a benchmarking loop and running the calculation code individually. We discovered that the difference in runtimes can be a few seconds for the $(128^2 \times 75)$ data size. But for the largest 2 datasets, the difference is insignificant. Instead, the alternative time in brackets in Table 6.11 is an estimate by calculating 4 feature values only and scaling it by 36.

For the second parallel strategy of taskfarming the 36 Gabor angle parameters to different threads described in section 4.6.3, the runtimes are in Table 6.12 where the times in brackets are obtained as part of a looping benchmark run. We do not run with all 16 available cores (except when using multi-threaded MKL) due to a lack of concurrency in the parallel scheme which is illustrated in the next section.

In both parallel schemes, the performance of the multi-threaded library is included that uses all 16 available cores. We obtain the runtime by specifying the MKL library to use 16 threads and time the Gabor feature vector calculation. The result is the last row in Tables 6.11 and 6.12.

The conclusions from the benchmarks on this compute-node are similar to those for the performance on *fermi-0* due to similar hardware but with faster CPUs, which can be seen by comparing the single thread runtimes. In addition:

- With 64GB of memory available, the taskfarm parallelisation strategy was able to run on 12 threads.

	Input Data Size (Time in seconds)			
# of Threads	$(64^2 \times 37)$	$(128^2 \times 75)$	$(256^2 \times 150)$	$(512^2 \times 300)$
1	8.62s (8.47s)	76.94s (77.24s)	595.82s (591.34s)	(6243.45s)
2	4.64s (4.08s)	38.50s (38.65s)	294.66s (291.26s)	3110.78s (3103.48s)
3	3.52s (3.04s)	26.74s (26.56s)	198.95s (198.08s)	2097.35s (2103.55s)
4	2.76s (2.35s)	20.15s (20.22s)	150.60s (149.25s)	1582.35s (1579.38s)
6	2.14s (1.73s)	14.31s (14.18s)	104.84s (104.43s)	1100.39s (1100.55s)
9	1.71s (1.40s)	10.61s (10.51s)	77.92s (77.60s)	820.92s (819.28s)
12	1.46s (1.14s)	8.56s (8.43s)	61.85s (61.54s)	640.26s (646.36s)
16*	3.77s	23.88s	177.30s	1457.26s

Table 6.12: Time taken to calculate all 144 Gabor feature values using the taskfarm parallel strategy using Octave on the *phi* node using the CPUs. The times in brackets are obtained from a single benchmarking code, the other times are obtained by timing the Gabor feature calculation code each time with a specified number of threads and a data input. *Code is compiled with the multi-threaded MKL library.

- A variation exists in obtaining runtimes individually and from a benchmarking code that reruns the Gabor feature calculation after incrementing the number of threads. This variation tends to be insignificant for cases with large data sizes.

6.3.2 Concurrency Analysis of Taskfarm Parallel method

We use the Intel VTune performance profiling tool provided by the Intel SDK to examine the concurrency of the taskfarm parallel strategy when 16 threads are used. For this purpose, we use a C program that calls the Gabor feature calculation with a random $(256^2 \times 150)$ array and the same Gabor filter parameters (6.1) used by the original program.

We run on the *phi* node:

```
$ amplxe-cl -collect concurrency \
             bin/gabor_3d_features_native_omp36_RUN
```

where `amplxe-cl` is the VTune profiler program without the GUI, the second argument asks to profile the program concurrency and the last argument is the C program. The following summary is provided:

```
Summary
-------
Average Concurrency:  12.919
Elapsed Time:         64.933
CPU Time:             828.239
Wait Time:            193.840
CPU Usage:            12.501
```

where we see that only about the equivalent of 12.9 cores were fully utilised. This is due to 16 not dividing into the 36 available tasks. Finally, Fig. 6.15 from the VTune

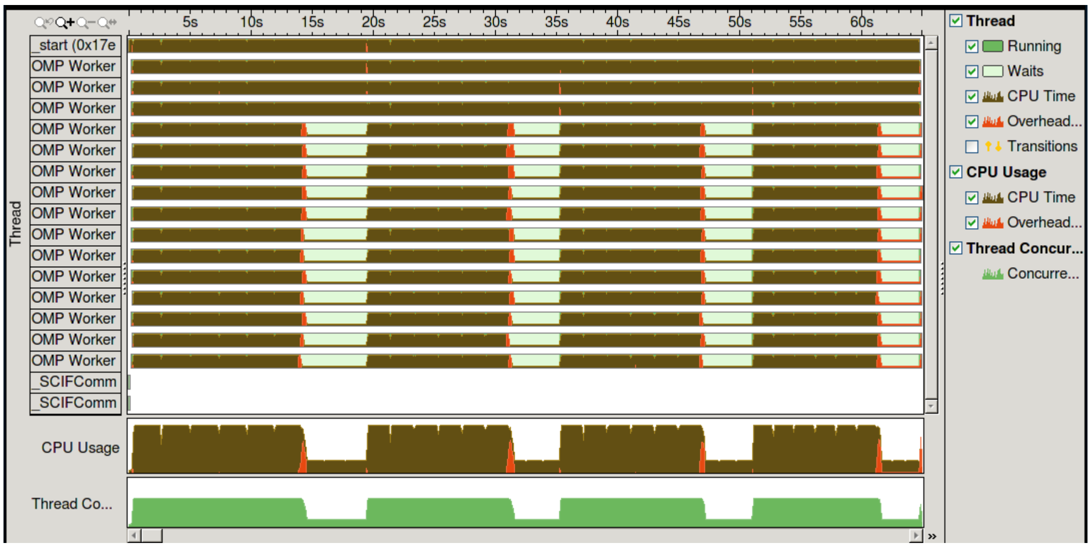


Figure 6.15: Concurrency graphic shown by the Intel VTune profiler. The profiled C program calculates the Gabor features vector using the taskfarm parallel strategy with 16 cores.

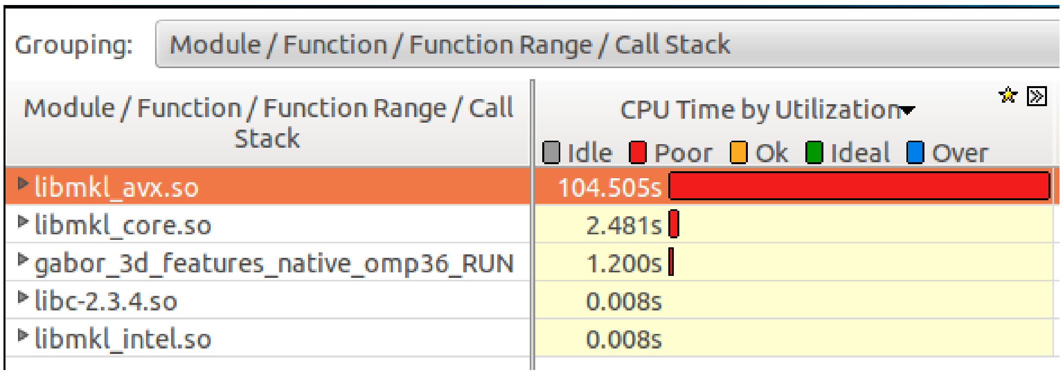
GUI shows the problem where at distinct stages, 4 threads continue to work whilst there are no more tasks to distribute to the others. This is the reason why upto 12 cores are selected when using the taskfarm parallelisation method. This suits the *fermi-0* compute-node where only 12 real cores are available but 4 cores are spared on the *phi* frontend-node where 16 are available.

6.3.3 Code Hotspot Analysis of Taskfarm Parallel method

By compiling with the debug `-g` flag, the VTune profiler can provide a basic code hotspots analysis of the same C program that we used for concurrency analysis. We run the program with a smaller data size and use a single thread. With a total runtime of 108.53 seconds[1], Fig. 6.16 shows the share of that runtime in various shared object files. This shows that nearly all the runtime involves the shared object `libmkl_avx.so` hence the bulk of the runtime is spent by the MKL library executing the convolution tasks.

Finally, Fig. 6.17 displays the CPU timings of the source code of the underlying function to calculate the Gabor features vector. This shows that the time taken to calculate intermediate results such as the computation of the energy norm and calculating the Gabor feature value at the end is of the order of milliseconds compared to the total runtime of about 108 seconds.

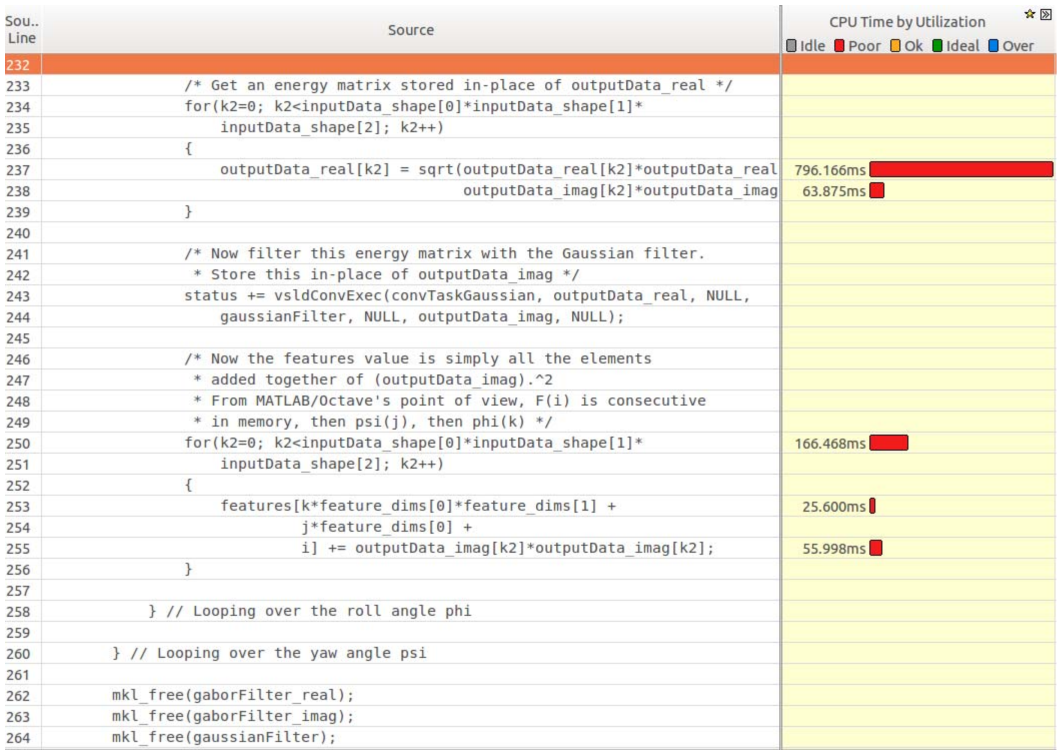
[1]The VTune profiler analysis was done on a laptop with an Intel Core i5-2467M at 1.6GHz. A copy of the Intel SDK was used under a non-commercial licence.



Grouping: Module / Function / Function Range / Call Stack

Module / Function / Function Range / Call Stack	CPU Time by Utilization (⬜ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)
libmkl_avx.so	104.505s
libmkl_core.so	2.481s
gabor_3d_features_native_omp36_RUN	1.200s
libc-2.3.4.so	0.008s
libmkl_intel.so	0.008s

Figure 6.16: Time spent in various shared objects. Our Gabor feature calculation code only takes 1.2s.



```
                                                                        CPU Time by Utilization
            /* Get an energy matrix stored in-place of outputData_real */
            for(k2=0; k2<inputData_shape[0]*inputData_shape[1]*
                inputData_shape[2]; k2++)
            {
                outputData_real[k2] = sqrt(outputData_real[k2]*outputData_real   796.166ms
                                           outputData_imag[k2]*outputData_imag    63.875ms
            }

            /* Now filter this energy matrix with the Gaussian filter.
             * Store this in-place of outputData_imag */
            status += vsldConvExec(convTaskGaussian, outputData_real, NULL,
                gaussianFilter, NULL, outputData_imag, NULL);

            /* Now the features value is simply all the elements
             * added together of (outputData_imag).^2
             * From MATLAB/Octave's point of view, F(i) is consecutive
             * in memory, then psi(j), then phi(k) */
            for(k2=0; k2<inputData_shape[0]*inputData_shape[1]*                 166.468ms
                inputData_shape[2]; k2++)
            {
                features[k*feature_dims[0]*feature_dims[1] +                      25.600ms
                         j*feature_dims[0] +
                         i] += outputData_imag[k2]*outputData_imag[k2];           55.998ms
            }

        } // Looping over the roll angle phi

    } // Looping over the yaw angle psi

    mkl_free(gaborFilter_real);
    mkl_free(gaborFilter_imag);
    mkl_free(gaussianFilter);
```

Figure 6.17: Gabor feature calculation source code together with the hotspot information.

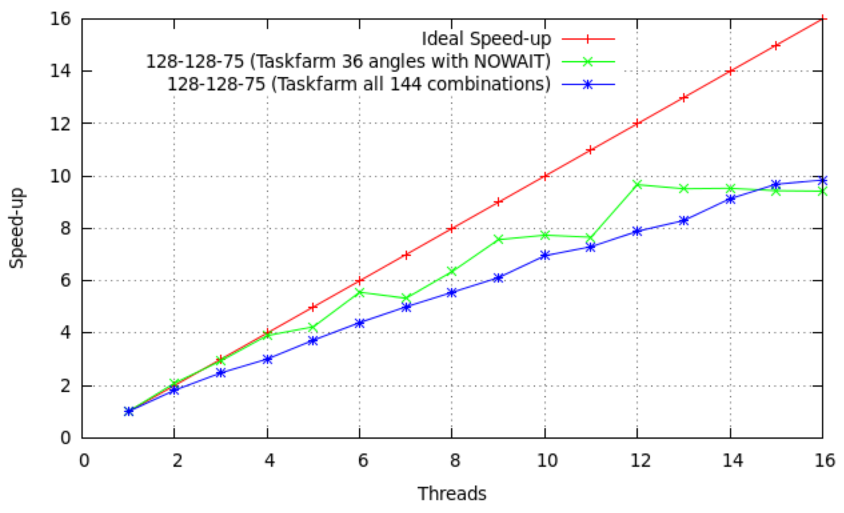


Figure 6.18: Comparison of speed-up between the two modified taskfarming parallel methods. One uses the NOWAIT clause whilst the other distributes all 144 Gabor filter parameters concurrently.

6.3.4 Increasing Concurrency of Taskfarm method

Given that our original taskfarm strategy suffered from reduced concurrency if the number of threads do not divide exactly into the 36 angle combinations available for each frequency, we try increasing the concurrency by:

- Adding `nowait` to the OpenMP directive `#pragma omp for collapse(2)` so any waiting threads can immediately proceed to the next frequency parameter.
- Taskfarming all 144 combinations of (6.1) in parallel, described in section 4.6.4

Figure 6.18 shows the result of attemping these two suggestions on a $128^2 \times 75$ array. Using `nowait` has no effect on the concurrency and the step behaviour is still obtained. Both schemes achieve 77.1 seconds for a single thread but the second method is only able to outperform the best time of the first one by 0.14 seconds and requiring 16 threads to achieve this. However, this is predicted to scale better beyond 36 threads.

6.4 Performance on Intel Xeon Phi

With many x86 cores available on the Intel Xeon Phi, we investigate the performance and scaling of both taskfarming parallel schemes described in sections 4.6.3 and 4.6.4, beyond the 12 or 16 threads available on the CPUs.

For this work, we use native C programs to execute the calculation of the Gabor feature vector using a randomly generated 3D array of double-precision numbers for the data but the Gabor feature parameters in 6.1 remain the same. Compilation for the Xeon Phi is achieved with the `-mmic` flag and the binaries are uploaded and executed on the card itself. To ensure use of separate cores by separate threads, the thread affinity scheme is set to scatter.

Finally, we demonstrate Octave calling MEX files that offload computations onto the Xeon Phi co-processors.

6.4.1 Taskfarming the 36 angle parameters

The concurrency of this parallel strategy is 36 so we test the performance using the Xeon Phi upto 36 cores. Similarly to what we did with the Xeon CPUs on the *phi* node in subsection 6.3.1, we obtain the runtimes first by using a benchmarking code that increments the number of threads in a loop and timing the calculation in the loop body. After obtaining the speed-up performance this way, we time the calculation code individually on a select number of threads in order to compare and confirm the benchmark estimation. We refer to this as a 'cold-start' so that the times obtained are not affected by any warm-up effects when timing the calculation in a loop body.Figure 6.19 shows the speed-up performance upto and including 36 threads. The jump in performance from 35 to 36 threads is expected where all threads work on all 36 available combinations per frequency parameter. For the larger dataset, the difference between the speed-up obtained by the two timing methods is minimal but the smaller dataset exhibits a ceiling in performance when running and timing the Gabor feature calculation code individually for 18 and 36 threads. This could indicate that the few Gabor feature calculations that each thread does is fast enough such that other overheads such as cold-cache misses and initial data movement now dominate the runtime.

Table 6.13 quotes the times to calculate the feature vector for 1 and 36 threads. By comparing with the times obtained with the Xeons, the best performance obtained on the Xeon Phi is less than that obtained on the Xeon CPUs due to lack of concurrency in the parallel method. In addition, we are unable to run the larger datasets due to insufficient memory. For example, an array of size $(256^2 \times 150)$ was limited to 9 threads in our experiments on the Xeon Phi.

# of Threads	$(64^2 \times 37)$	$(128^2 \times 75)$
1	92.85s	993.60s
36	3.10s (7.76s)	33.23s (34.05s)

Table 6.13: Calculation times for the Intel Xeon Phi for 1 and 36 running threads for the 2 data sizes used. The times in brackets are the 'cold-start' times.

6.4.2 Taskfarming all 144 parameters concurrently

Figure 6.20 shows the speed-up performance on the Xeon Phi upto 144 threads using this parallel method described in subsection 4.6.4. Similarly with the previous taskfarm

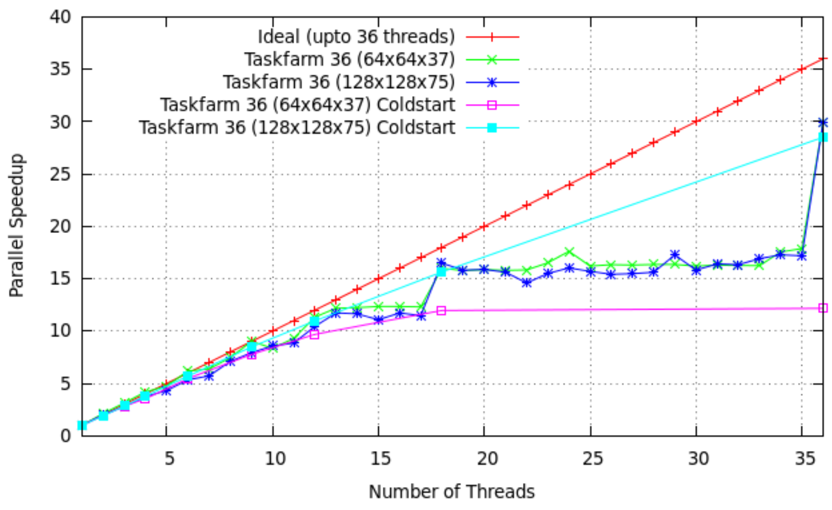


Figure 6.19: Speed-up performance on the Xeon Phi using the parallel strategy of distributing the 36 different angle parameters. Times were obtained in two different ways (looping benchmark vs. individual 'cold-start' runs) and the sizes of the datasets used are shown in brackets.

parallel method, the benchmarking runs overestimate the performance of the Gabor feature calculation where a user would run the code just once to calculate the features for a single image. Table 6.14 provides metrics for the best times obtained under the 'cold-start' method for both data sets.

Image Size	Best time (s)	# of Threads	Speed-up	Par. Efficiency
$(64^2 \times 37)$	5.12	33	18.0	0.55
$(128^2 \times 75)$	21.55	54	40.5	0.75

Table 6.14: Metrics for the best times obtained on the Xeon Phi using the parallel strategy of distributing all 144 parameters concurrently under the 'cold-start' method.

Highlights of this work in this subsection are:

- For both data sizes, distributing all 144 parameters increases the scalability of the calculation. However, both cases level off after a number of threads that is much lower than 144. The smallest data size attains maximum performance after about 33 threads. For the larger data, levelling off occurs after about 60 threads which equals the core count. This would indicate a saturation in memory bandwidth because all the internal buffers used for memory requests are in use when all 60 cores are used to maximize memory bandwidth [17].
- The 8GB of on-board memory is insufficient with the $(128^2 \times 75)$ data beyond 80 threads.
- The best times obtained on the Xeon Phi are slower than what is possible on the Xeon CPUs.

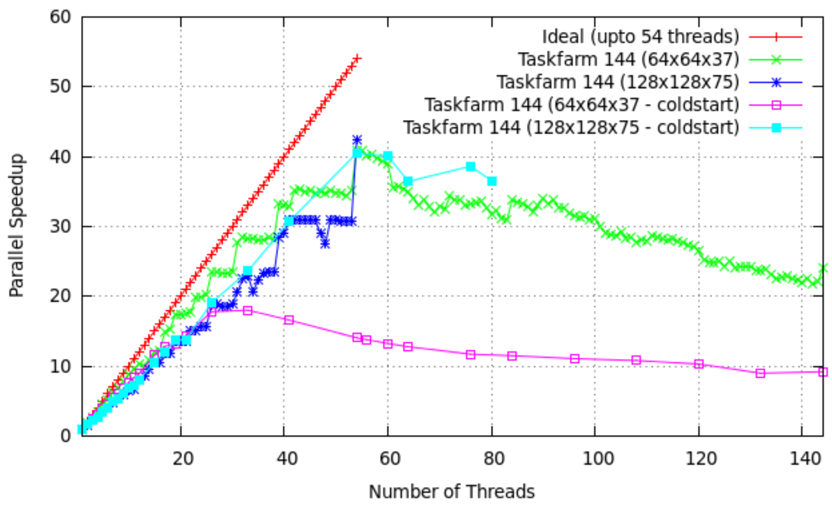


Figure 6.20: Speed-up performance on the Xeon Phi using the parallel strategy of distributing all 144 parameters concurrently. Times were obtained in the same way as for Fig. 6.19.

- The application is not able to fully utilise the computational performance of the Xeon Phi due to the application unable to scale to at least 120 threads. Each core is expected to run at least two threads to maximize floating point performance [17].

6.4.3 Using Multi-Threaded MKL

To overcome the limit of available on-board memory, we can use the multi-threaded version of MKL to automatically parallelise the calculation when the convolution task objects are executed. The serial code outlined in section 4.6.2 is compiled with the parallel MKL libraries and the times obtained on the Xeon Phi are in Table 6.15.

# of Threads	$(64^2 \times 37)$	$(128^2 \times 75)$	$(256^2 \times 150)$	$(512^2 \times 300)$
60	25.7287s	146.077s	1064.71s	8177.06s
120	19.8848s	114.736s	1035.24s	7739.33s
180	19.1939s	119.266s	1009.95s	7625.41s
240	19.944s	118.885s	1019.5s	7650.44s

Table 6.15: Runtime of the Gabor feature calculation on the Intel Xeon Phi using multi-threaded MKL on upto 240 threads.

Comparing with the single thread times in Table 6.13, some speed-up is offered by the MKL library. All four data sizes were possible to run, however performance is worse than a single thread on the host CPU in Table 6.11.

6.4.4 Offloading from Octave

A working setup is demonstrated where Octave calls a MEX file that attempts to offload the calculation of the Gabor feature vector onto a Xeon Phi co-processor.

The offloading capability is provided by offload pragmas inserted into the source code in the same fashion as OpenMP directives. These include controlling the data flow between the host and co-processor to provide optimisation opportunities for copying data.

The offload directives within the code responsible for Gabor feature calculation is:

```
  /* Begin offload onto the Xeon Phi */
#pragma offload target (mic) \
                  in(frequency :length(# of frequencies))   \
                  in(yaw angle :length(# of yaw angles))    \
                  in(roll angle:length(# of roll angles))   \
                  in(Data      :length(size of Data))       \
                  in(shape of Data:length(3))               \
                  inout(Gabor feature vector:length(total of
                                           parameters combinations))
{
   Set number of OpenMP threads

  (OpenMP directives)
  {
     code to calculate Gabor feature vector
  }
}
```

so the data to be convoluted, its shape and the feature parameters are copied onto the card and the Gabor feature vector is passed in and out at the end (though this could be optimised by creating the vector on the card itself). When a calculation is currently offloaded onto the co-processor, a line similar to the one below appears in the output of `top` running on the card through an ssh command-line:

```
5361 3442 micuser S 1318m 17.1 9142.2 /tmp/coi_procs/1/5361/offload_main
```

To enforce scatter placement of threads when jobs are offloaded, the following environment variables ensure that the Xeon Phi uses the scatter placement:

```
$ export MIC_ENV_PREFIX   = MIC
$ export MIC_KMP_AFFINITY = scatter
```

Figure 6.21 shows the calculation times by offloading and by native execution on the card itself. Because the data size to transfer is at most under 10MB, this does not affect the runtime significantly and we come across an occasion of the offload run outperforming the native run.

Below is an example offload report for the case of 9 threads on a $(128^2 \times 75)$-sized data (export OFFLOAD_REPORT=2):

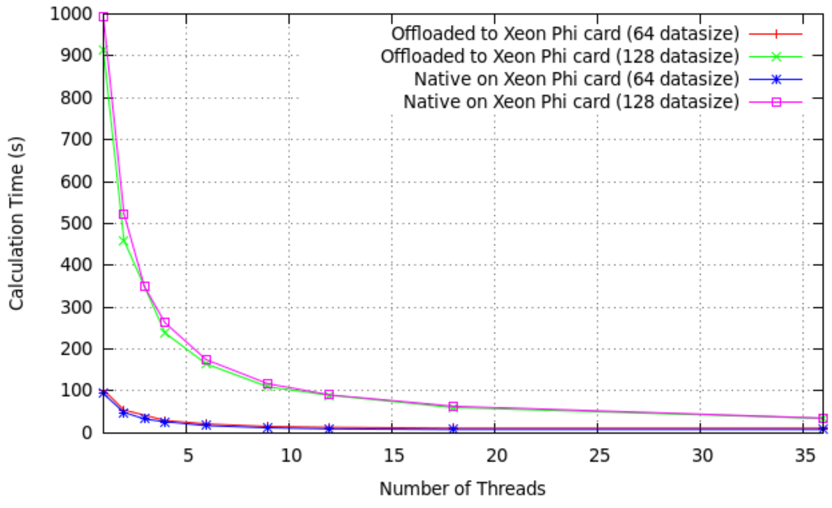


Figure 6.21: Gabor feature vector calculation on both the Xeon Phi natively and on Octave offloading the calculation onto the card.

```
[Offload] [MIC 0] [File]            src/gabor_3d_features_native_xphi.c
[Offload] [MIC 0] [Line]            44
[Offload] [MIC 0] [Tag]             Tag 0
[Offload] [HOST]  [Tag 0] [CPU Time]        108.267035(seconds)
[Offload] [MIC 0] [Tag 0] [CPU->MIC Data]   9831744 (bytes)
[Offload] [MIC 0] [Tag 0] [MIC Time]        104.765726(seconds)
[Offload] [MIC 0] [Tag 0] [MIC->CPU Data]   1160 (bytes)
```

so the calculation completes on the card in 108 seconds which is faster than a native time of 117s on one occasion.

To summarise, we have shown here a setup where it is possible to offload computations onto the Xeon Phi from the Octave programming environment. However, the calculations on the card itself remain to be optimised as the performance attainable is still below what is possible on a multicore CPU node.

6.5 Chapter summary

We have demonstrated performance improvements over the original MATLAB application by linking the MATLAB code to C functions that use the MKL library and OpenMP for parallelisation.

Two parallel strategies were employed to achieve such speed-up. The first one is dividing the data to be correlated into smaller, equal subdomains amongst the running threads. The second involves each thread performing the equivalent convolution on the whole data for particular combinations of parameters for the filter kernel.

The original MATLAB application computed the correlation of the data and filters by direct computation of the correlation formula. Through use of the equivalent routines in the MKL libraries using FFTs in addition to parallelisation, we have achieved a speed-up of between 42 and 168 times over the original application. The second parallel strategy provided the lowest runtimes but was found to have a large memory footprint such that the largest data example was unable to run on all available cores.

Due to a lack of concurrency in the second parallel strategy, we developed a modified version to expose more available parallelism and explored the scalability of both versions on an Intel Xeon Phi with many more cores available. The speed-up performance was not sufficient to make full use of the card's capabilities and the runtimes obtained were longer than what was achieved on the multi-core CPU nodes.

Finally, we demonstrated a working framework where GNU Octave is able to offload the computations onto a Xeon Phi co-processor. This shows the possibility of using optimised code for the Xeon Phi from the Octave programming environment in the future.

Chapter 7

Summary and Conclusions

The project's aims were to accelerate a medical imaging application provided by Western General Hospital. This application helps to predict the onset of lung-pneumonitis in lung cancer patients after radiotherapy treatment. This is achieved by calculating statistical and numerical features of the textures on the CT scan images of the patient's lungs. This calculated texture data is fed into a system that predicts whether the patient will suffer the illness.

The application is originally written in MATLAB that calculates the texture features of three-dimensional image data. A preliminary study was carried out to determine the most expensive computational routines of the program and concluded that the texture feature calculations involving the correlation of the data with the Gabor and Gaussian filters employed an algorithm of computational complexity that had $\mathcal{O}(N^3K^3)$ scaling.

The project's focus was to accelerate the application using the Intel Parallel Studio suite of tools that can make use of multicore CPU hardware and many-core CPU coprocessors using the same code developed in C. Due to the different programming languages employed between the Intel SDK and the original application in MATLAB, the most expensive computational routine required porting over to C code that was then callable by the application running in MATLAB. The project was successful in developing C code using OpenMP for multi-threading and Intel MKL routines for computation that can be utilised by the MATLAB programming environment.

We developed C functions using the Intel Math Kernel Libraries that provided correlation and convolution results equivalent to those returned by MATLAB. These routines by MKL can either compute the result by directly applying the correlation/convolution formula directly or by using Fast Fourier Transform algorithms.

We have found that, by using the MKL routines in FFT mode, a speed-up over the original application was already possible before any parallelisation was employed. The results show that the original application could be speeded up between 42 and 50 times on single threads simply by switching to FFTs algorithms that have less computational complexity than by computing the correlation formula directly.

For parallelisation onto multicore CPUs, we used OpenMP to multi-thread the computational routines involving Gabor feature texture calculation. Two parallel strategies were employed:

- Parallelisation of the correlation routine alone by dividing the data into equal subdomains onto different threads.
- Taskfarming the different Gabor filter parameters involved in the Gabor feature texture calculation onto multiple threads. Each thread computes the Gabor feature texture value on the whole data for a particular combination of the given parameters.

Form the benchmarking results on the first parallel strategy, the speed-up attained on multi-core CPUs showed good scaling on with parallel efficiency not dropping below 0.85 for evenly distributed workloads. This is for computing the correlation directly (with a fixed filter size) and was found to be 11 to 14 times slower than by using FFTs in single threaded runs, depending on the data size. When using FFT mode, the parallel efficiency drops to 0.5 for the largest two data sizes tested but for the smallest ones, the performance levels off after a few threads due to the correlation calculation completing in the order of milliseconds for single threads.

By applying both parallelisation strategies on the original MATLAB application, we have successfully reduced the runtime of the original application even further through the use of multicore CPUs on compute-nodes. The fastest runtimes were achieved using the taskfarming parallelisation method due to much of the intermediate calculations performed in C rather than in MATLAB. In general, improved runtimes were such that the Gabor feature calculation part of the total runtime is less than half. In terms of the order of the runtime compared to that for the original code:

- For the two smallest data sizes (64-64-47 & 128-128-75), we reduce the runtime to the order of a few second instead of hundreds of seconds.
- For the next size (256-256-150), we achieve runtimes of the order of minutes instead of hours.
- For the largest size (512-512-300), the computation takes between 10 and 20 minutes instead of days.

The best case for the improvement in application runtime on the node which has MATLAB installed is a 168 speed-up over the original time using all 12 cores on a data of size (256-256-150). However, one drawback is the large memory footprint to achieve this which is on the order of gigabytes. It was not possible to use all available CPU cores when computing the texture features of the largest data sample due to 48GB being insufficient.

Using the first parallelisation strategy resulted in slightly lower performance than the second one. But, because the single correlation routine is replaced, this maintains the programming flexibility of being able to easily modify other code in MATLAB that does not involve the correlation of the data, such as the intermediate calculations involved in between the correlation calls.

Compared to the two parallel strategies developed, the performance of the multi-threaded MKL library is better for the two smallest data sizes tested. But we obtain better performance for the larger two sizes.

With the application speed-up obtained, the error analysis between our solutions and the original indicates a maximum relative error of order $\mathcal{O}(10^{-12})$ amongst all the calculated Gabor feature values, which compares reasonably well to the double-precision limit on computers which is 16 significant figures. But the absolute difference is of the order $\mathcal{O}(10^{23})$ and it remains to be seen if these differences in values results in a different qualitative outcome.

By using the Intel VTune profiler on the code using the taskfarming parallel strategy but running with one thread, we conclude that the majority of the runtime is spent within the MKL library routines calculating the convolution or correlation. In the case presented, 1.2 seconds was spent in other parts of the function as opposed to 104 seconds in the MKL shared library object performing calculations using the AVX vector unit on the CPU. However, a concurrency analysis indicates that for a certain number of threads, one or more are idle for periods of time due to an uneven distribution of available work. In this case, only a total of 36 tasks are concurrently available and we have demonstrated that with 16 running threads, four threads will idle after everyone completes two tasks each.

The concurrency of the taskfarming method was increased by making all 144 Gabor filter parameter combinations available at the same time for threads to take, as opposed to the 36 different pairs of angles only. This version did not help in increasing performance on the CPUs due to this version unable to reuse the task objects because one of the Gabor filter parameters can change the size of the filter - task objects can be reused with different filters if they are the same size. Despite this, the scaling of this version indicates that this would be suitable for many more processing cores than the 12 or 16 cores available on the *fermi-0* or *phi* compute-nodes.

Both versions of the taskfarm parallel strategy were then tested on the Intel Xeon Phi co-processor to investigate the scalability beyond the 16 processor cores available. The 8GB of on-board memory was a problem such that only the two smallest data sizes can be realistically tested. But even with these images, the speed-up measured shows that, in general, the performance levels off before reaching the maximum concurrency possible. The one exception is the case of the larger size of the two using the first version with a maximum possible concurrency of 36, where the speed-up and parallel efficiency were 29.1 and 0.81 respectively using 36 threads. But in all cases, the runtime to calculate the Gabor feature values were longer than those achieved on the CPUs on the compute-nodes. This leads to the conclusion that the convolution and correlation routines in the MKL library do not provide performance as optimised as that for regular CPUs.

Using the multi-threaded MKL library allows a workaround of the on-board memory limit on the Intel Xeon Phi package. However, performance is no better than a single Xeon CPU in most cases. This shows that a manual parallel implementation is required to achieve optimal performance

The levelling of the speed-up on the Xeon Phi concludes that the strategy of each thread computing the convolution on the same data with different filters leads to a performance saturation beyond a certain number of threads. In one case, the performance starts decreasing after 60 threads are used, indicating a memory bandwidth saturation as all 60 cores are in use (using the scatter thread placement scheme), thus using all available internal memory buffers of the many-core chip.

Finally, we have shown the feasibility of GNU Octave calling compiled C functions that offload onto the Xeon Phi. Hence any future code development for optimal performance on the Xeon Phi can be taken advantage of from the Octave programming interface and likely so in the case of MATLAB.

Final Words

We have successfully accelerated the original medical imaging application in MATLAB by linking with it with code developed using the Intel SDK and parallelised using OpenMP. The reduction in runtimes achieved over the original application were substantial due to the ability of the called functions to perform the convolution and correlation using FFT algorithms.

Application speed-up was successful on multi-core CPUs. However, running the same code on the Xeon Phi co-processor resulted in less performance being obtained than that for Xeon CPUs and the on-board memory limited the size of the data to be processed.

7.1 Future Work

On a high level view, the two main areas of improvement are a reduction of memory usage and increasing performance on the Intel Xeon Phi co-processor.

The following list now looks at particular areas for future work:

- **Use of FFT libraries** Instead of using the convolution and correlation routines from Intel MKL in FFT mode, one can use a variety of FFT libraries directly to develop a parallel convolution solver. This would be useful for an alternative implementation of the Gabor feature calculations, especially tuned for the Xeon Phi as the correlation algorithms involved can be optimised to suit the Xeon Phi architecture. This would be similar to past work where a FFT-based routine was developed specifically for use by PowerPC processors on an IBM Blue Gene [28].

- **Further Mathematical/Algorithmic Improvements** Currently, calculating a Gabor feature value involves correlations of the real and imaginary parts of the Gabor filter separately. We should investigate using the complex versions of the correlation routines where we correlate the real image data with the complex Gabor filter kernel and the subsequent intermediate calculations can be done on this new result. One of the intermediate calculations is an energy matrix and we should investigate if it is possible to perform this calculation in the Fourier domain to save the need for a Fourier and an inverse Fourier transform stage. Finally, modifying the workflow to use in-place FFTs can help reduce memory usage as intermediate calculations are stored in-place.

- **Use of Other Compilers and Libraries** The project exclusively used the Intel compilers and MKL libraries, which limits its use on other systems as the Intel SDK is not free and users may be willing to trade some performance for the benefit of free compilers and libraries. Examples would be use of GCC and free convolution/correlation libraries to increase portability.

- **Different Domain Decompositions** Instead of dividing the data into equal subdomains, we look at alternatives such as pencil and slab that may be more suitable for the FFT algorithm that the convolution routines use internally. Alternatively, specific subdomain lengths, including the possibly of padding with zeros, can be investigated such that the lengths are suitable for FFT algorithms.

- **Multi-node use** After any improvements in the program's memory footprint, the taskfarming scheme developed here can be extended so a subset of the Gabor filter parameters can be distributed amongst nodes using a suitable framework such as message-passing programming. Alternatively, multi Xeon Phi co-processors or worksharing amongst Xeon processors and Xeon Phi co-processors could be investigated after developing code for optimal performance on the Intel Xeon Phi.

The first two items described above have enough scope to improve the two main areas needing improvement, namely high memory usage and performance on the Intel Xeon Phi. The fourth item would help optimise the general performance of FFT algorithms.

Appendix A

Known Software Issues

A.1 Testing errata on different GNU Octave versions

The testing scripts developed to run on Octave will only work with version v3.6.4 of Octave: the starting point for the ‘same’ convolution/correlation is the same as on MATLAB R2012a. For the earlier version of Octave v3.4.3, installed on the *phi* node, the ‘same’ convolution starts at a different position so the tests checking for equivalent convolution/correlation answers between Octave and our MKL-based routines will fail.

Bibliography

[1] M. Bull, X. Guo and I Liabotis, Applications and user requirements for Tier-0 systems, http://www.prace-project.eu/IMG/pdf/D7-4-1.pdf, last accessed 21/08/2013.

[2] Top500 November 2011 Poster, http://s.top500.org/static/lists/2011/11/TOP500 _201111_Poster.pdf, last accessed 06/04/2013.

[3] How to make best use of the AMD Interlagos processor, Numerical Algorithms Group, Nov 2011, www.hector.ac.uk/cse/reports/interlagos_whitepaper.pdf, last accessed 21-08-2013.

[4] IBM Cell Broadband Engine, https://www-01.ibm.com/chips/techlib/techlib.nsf/ products/Cell_Broadband_Engine, last accessed 21-08-2013.

[5] The OpenACC Application Programming Interface, http://www.openacc.org/ sites/default/files/OpenACC.1.0_0.pdf, last accessed 21-08-2013.

[6] OpenCL, http://www.khronos.org/opencl, last accessed 21-08-2013.

[7] D Montgomery, K Cheng, Y Feng, D B McLaren, S C Erridge, S McLauglin, S Campbell, and W H Nailon, Predicting the Occurrence of Radiation Induced Pneumonitis by Texture Analysis of CT Images from Lung Cancer Patients, preprint (2012).

[8] J. Low, Medical Image Processing on Intel Parallel Frameworks, Project Preparation Report, MSc in High Performance Computing, 2013.

[9] About lung cancer, Cancer Research UK, http://www.cancerresearchuk.org/ cancer-help/type/lung-cancer, last accessed 21-08-2013.

[10] D. G. Dessavre, Using High Performance Computing to Improve Image Guided Cancer Treatment, MSc Dissertation (2012), The University of Edinburgh.

[11] N. Petkov and M.B. Wieling, Gabor filter for image processing and computer vision (web application), Department of Computing Science, Intelligent Systems, University of Groningen, http://matlabserver.cs.rug.nl/edgedetectionweb/ web/index.html, last accessed on 21-08-2013.

[12] C. Lee, S. Chen, H. Tsai, P. Chung and Y. Chiang, Discrimination of Liver Diseases from CT Images Based on Gabor Filters, Proceedings of the

19th IEEE Symposium on Computer-Based Medical Systems (CBMS'06), doi:10.1109/CBMS.2006.77

[13] L. Shen and L. Bai, 3D Gabor wavelets for evaluating SPM normalization algorithm, Medical Image Analysis 12 (2008) 375-383.

[14] Z. Qian, D.N. Metaxas, L. Axel, Extraction and Tracking of MRI Tagging Sheets Using a 3D Gabor Filter Bank, Conf Proc IEEE Eng Med Biol Soc., 1:711-4 (2006).

[15] Optimization Notice, Intel Inc., *http://software.intel.com/en-us/articles/optimization-notice#opt-en*

[16] OpenMP Application Program Interface, Version 4.0 - July 2013, available at http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[17] J. Jeffers and J. Reinders, Intel Xeon Phi Coprocessor High Performance Programming, Elsevier Science & Technology Books, 2013.

[18] D. Molka, D. Hackenberg, R. Schone and M.S. Muller, Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System, 2009 18th International Conference on Parallel Architectures and Compilation Techniques, doi:10.1109/PACT.2009.22

[19] GNU Octave, http://www.gnu.org/software/octave

[20] G. Chrysos, Intel Xeon Phi coprocessor (codename Knights Corner), presentation at Hot Chips, August 28 2012, available at http://newsroom.intel.com/servlet/JiveServlet/download/38-11511/Intel_Xeon_Phi_Hotchips_architecture_presentation.pdf, last accessed 20-08-2013.

[21] V. Podlozhnyuk, FFT-based 2D convolution, June 2007, NVIDIA Corp, http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/convolutionFFT2D/doc/convolutionFFT2D.pdf, last accessed 30-05-2013.

[22] J. Fix, "Efficient convolution using the Fast Fourier Transform, Application in C++", http://jeremy.fix.free.fr/IMG/pdf/fftconvolution.pdf, last accessed 02-06-2013.

[23] Intel Math Kernel Library Reference Manual, MKL 11.0 update 5, available at http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mklman/mklman.pdf, last accessed 20-08-2013.

[24] Introducing MEX-Files, The MathWorks website, http://www.mathworks.co.uk/help/matlab/matlab_external/introducing-mex-files.html, last accessed 05/04/2013.

[25] OpenMPI: A High Performance Message Passing Library, http://www.open-mpi.org, last accessed 06/06/2013.

[26] FFT length and layout advisor, Intel Developer Zone article, http://software.intel.com/en-us/articles/fft-length-and-layout-advisor, last accessed 21-08-2013.

[27] FFT Benchmark Results, http://www.fftw.org/speed, last accessed 20-08-2013.

[28] A. Nukada, Y. Hourai, A. Nishida and Y. AkiyamaHigh, Performance 3D Convolution for Protein Docking on IBM Blue Gene, Parallel and Distributed Processing and Applications, Lecture Notes in Computer Science, Vol 4742, 2007, pp 958-969.