12-13-2013 12:00 AM

Performance Comparison of 3D Sinc Interpolation for fMRI Motion Correction by Language of Implementation and Hardware Platform

Andrew B. Kope, *The University of Western Ontario*

Supervisor: Mark Daley, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

PERFORMANCE COMPARISON OF 3D SINC INTERPOLATION FOR fMRI MOTION CORRECTION BY LANGUAGE OF IMPLEMENTATION AND HARDWARE PLATFORM

by

Andrew Kope

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

Substantial effort is devoted to improving neuroimaging data processing; this effort however, is typically from the algorithmic perspective only. I demonstrate that substantive running time performance improvements to neuroscientific data processing algorithms can be realized by considering their implementation. Focusing specifically on 3D sinc interpolation, an algorithm used for processing functional magnetic resonance imaging (fMRI) data, I compare the performance of Python, C and OpenCL implementations of this algorithm across multiple hardware platforms. I also benchmark the performance of a novel implementation of 3D sinc interpolation on a field programmable gate array (FPGA). Together, these comparisons demonstrate that the performance of a neuroimaging data processing algorithm is significantly impacted by its implementation. I also present a case study demonstrating the practical benefits of improving a neuroscientific data processing algorithm's implementation, then conclude by addressing threats to the validity of the study and discussing future directions.

Keywords

Functional Magnetic Resonance Imaging, 3D Sinc Interpolation, Benchmarking, Parallel Programming, OpenCL, Graphics Processing Unit, Field Programmable Gate Array, Motion Correction, fMRI, GPU, FPGA.

Acknowledgments

I would like to thank my supervisor Mark Daley for his guidance and feedback throughout my master's degree, and acknowledge Conor Wild for his assistance with the evaluation of my implementations of 3D sinc interpolation in the context of a practical motion correction algorithm.

Table of Contents

List of Tables

List of Figures

List of Appendices

Preface

Within neuroscience, there is substantial effort devoted by researchers to the improvement of neuroimaging data processing. This effort however, is typically only from the algorithmic perspective with little attention paid to the hardware platform or programming language used for the implementation of any particular algorithm. In my thesis, I demonstrate that substantive improvements to the running time performance of a neuroscientific data processing algorithm can be realized by giving consideration to its implementation.

I begin in Chapter 1 with a review of the physics underlying the functional magnetic resonance imaging (fMRI) technique, followed by an overview of fMRI motion correction. Motion correction is a preprocessing algorithm used to correct errors in fMRI data caused by motion in the subject of the scan.

In Chapter 2, I provide a survey of several different interpolation methods used in the performance of fMRI motion correction. For each interpolation method, I provide an example of a popular software package which employs it in its motion correction utility.

In Chapter 3, I focus my discussion on the 3D sinc interpolation method, providing a detailed description of the algorithm alongside a discussion of both serial and parallel conceptualizations of it. 3D sinc interpolation provides highly accurate interpolations of fMRI data, however its substantial running time limits its applicability; for these reasons, I chose 3D sinc interpolation for my demonstration of the performance improvements which can be realized by considering the hardware platform and programming language used to implement an algorithm.

In Chapter 4, I quantify these improvements by benchmarking the performance of the 3D sinc interpolation algorithm in Python, C, OpenCL (Open Computing Language) for CPU/GPU across several different test bed hardware platforms. The benchmarking results demonstrate that parallel implementations can greatly improve the running time performance of the 3D sinc interpolation algorithm.

In Chapter 5, I describe a novel implementation of 3D sinc interpolation on a field programmable gate array.

In Chapter 6, I present a case study demonstrating the ‘real world’ benefits of these performance improvements by showing the corresponding decrease in running time of an algorithm for robust fMRI motion correction developed by my collaborators. I first explain their algorithm in detail and then show the results of performance benchmarking demonstrating these benefits.

In Chapter 7 I conclude by addressing threats to the validity of my results and discussing future directions for this research.

Chapter 1

1 Functional Magnetic Resonance Imaging

Functional Magnetic Resonance Imaging (fMRI) is one of the foremost neuroscientific imaging modalities in use today. This imaging technique uses the natural magnetic properties of the human body to capture both structural images of the brain's anatomy and functional images of its activity. fMRI is however vulnerable to errors introduced by even slight motion in the subject of the scan; as such, motion correction algorithms have been developed to correct for these errors. In this chapter a brief explanation of the physics behind Magnetic Resonance Imaging is provided alongside an outline of the structure of a general motion correction algorithm and an explanation of related concepts and terminology.

1.1 Magnetic Resonance Imaging Physics

Magnetic resonance imaging (MRI) is a medical imaging technique which works by taking advantage of the nuclear magnetic resonance (NMR) of the Hydrogen atoms inside the human body. During an MRI scan, the subject of the scan is placed into a very strong magnetic field imposed by a superconducting electromagnet which aligns the magnetization of the Hydrogen nuclei (protons) in the water molecules within her body. Secondary magnetic fields are then repeatedly applied to the subject by the scanner's magnetic gradient coils, combined with strong radio frequency (RF) pulses which alter the alignment of these magnetized protons; when the protons in the subject's tissues return to their previous alignment a response radio frequency signal detectable by the scanner's RF receivers is produced. These raw RF data recorded by the scanner’s receivers are in k-space. k-space data exist in the spatial frequency domain and so are not readily human readable. These data are converted into image space using an inverse Fourier transform after image acquisition in complete (Twieg, 1983).

With the ability to detect both the location and type of the tissues in the subject's body in three dimensions, MRI scanners can be used to capture 3D images of the complex tissue structures within the human brain (Figure 1). A 3D MRI image of the brain is referred to

as a *volume*. Each volume is composed of a stack of 2D images called *slices*. The minimum discrete data unit within each MRI volume is referred to as a *voxel* (akin to a pixel in a 2D picture); these voxels are cubes containing only intensity data (i.e. the voxels appear only in gray scale).

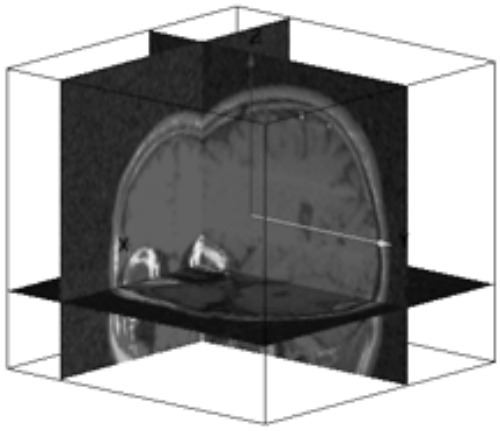

Figure 1: MRI brain image.

Functional magnetic resonance imaging uses the same physical principles underlying structural MRI brain imaging to measure subjects' brain activity over time (hence the term functional). The human body uses hemoglobin, the primary constituent of red blood cells, to transport oxygen within the blood stream. Importantly, oxygenated hemoglobin is unaffected by magnetic fields however deoxygenated hemoglobin is paramagnetic and distorts magnetic fields. fMRI scanning measures brain activity by relying on the positive correlation between brain activity and the presence of oxygenated hemoglobin within the brain (Ogawa et al., 1992).

When an area of the brain is active, after approximately two seconds there is a significant increase in the amount of oxygenated hemoglobin in the blood in that brain area, displacing deoxygenated hemoglobin. Because this oxygenated hemoglobin is unaffected by magnetic fields, the response RF signal returned to the fMRI scanner is stronger when there is more brain activity and therefore more oxygenated hemoglobin in that brain tissue; this is called the blood oxygen level dependent (BOLD) signal. The peak BOLD response to neuronal activation occurs approximately 5-6 seconds after the onset of

activity. In order to track neuronal activation in brain tissue over time, an entire brain volume is captured approximately every 1-2 seconds with a resolution of approximately 27mm3/voxel in 3T (tesla) fMRI research scanners. The capture of a series of many fMRI brain volumes over time is referred to as a *scanning session*, with the complete dataset from one scanning session referred to as a *time series*. A time series might contain for example 800 volumes, if a brain volume is captured every 1.5 seconds and the scanning session lasts for 20 minutes. The time between the capture of each volume is called the time of repetition (TR) or sampling time of the scanning session; increasing the resolution of a scan is traded for an increase in TR. It is also important to note that fMRI scanners vary in the strength of the magnetic fields used, with 1.5-7T fields being common among modern research scanners. With a higher magnetic field strength, the scanner can capture brain volumes faster (i.e. with a smaller TR) or at a higher resolution.

Before fMRI brain scan data can be reviewed or analyzed, they must first be subjected to several preprocessing steps. After the initial Fourier transform to convert raw k-space scanner data to image space, operations to remove anatomical artifacts, remove scanner noise and improve neuronal activation detection are typically applied (for more information see Strother's 2006 review of BOLD fMRI preprocessing pipelines). The focus of the present chapter is on the processing of fMRI brain scan data to compensate for slight involuntary head movements in the subject during the scanning session, referred to as motion correction.

1.2 fMRI Motion Correction: Overview

Motion correction was first introduced by Jiang et al. (1995) to "reduce the effect of subject motion during the acquisition of image data in order to differentiate true brain activation from artifactual signal changes due to subject motion" (p. 224). Because the observable signal changes in fMRI scanning are small, even with head movements of less than 1mm, spurious clusters of task-related brain activation can appear (Field, Yen, Burdette & Elster, 2000). For example, if two neighbouring voxels differ in intensity by 20%, then a motion of 10% of a voxel dimension can result in a 2% signal change, comparable to the BOLD signal in a 1.5T fMRI scanner (Bandettini et al., 1992). In the past, motion correction has received some criticism for potentially introducing spurious

activation artifacts itself. Freire and Mangin (2000) for example argued that some motion correction algorithms will actively misalign motion-free fMRI data if there is an unusual distribution of background noise or neuronal activation. Despite these criticisms however, motion correction is generally accepted as an integral component of any fMRI preprocessing pipeline (Lemieux et al., 2007; Lund et al., 2005).

In actuality, motion correction is a special case of image registration. As it pertains to MRI and fMRI data, "to register two images means to align them, so that common features overlap" (Kostelec & Periaswamy, 2003, p. 161). One common application of image registration for fMRI data is aligning different subjects' brain regions to a common anatomical template to enable comparisons between subjects in a study. In the context of motion correction, all of the volumes in one subject's scanning session are aligned to a common positional template to enable comparisons between volumes across a scanning session.

1.3 fMRI Motion Correction: General Algorithm

In the general case, the image alignment responsible for correcting motion in fMRI is done by the iterative performance of three major steps on each volume in a scanning session time series:

1. Determination of the difference (error) between the current image and a template image (e.g. the first image in a scanning session) using a cost function.

2. Application of an optimization algorithm to determine a spatial transformation to move the current image closer to the template image.

3. Interpolation of the scan data based on the spatial transformation from step two to create a new current image for the next iteration of the algorithm.

These three steps will continue until the error determined in step one is below a threshold determined by the optimization algorithm in step two. The error function in step one, the optimization algorithm in step two and the interpolation method in step three are each dependent on the motion correction algorithm used. In addition to these three algorithmic

dimensions, outlined below are several other dimensions along which different motion correction algorithms can vary; examples of specific motion correction algorithms varying along these dimensions are provided in the next chapter.

First, a motion correction algorithm can work either *online* or *offline*. An online algorithm corrects for subject motion while the scan is in progress and thus must be able to correct the motion of one volume in less time than the TR of the scanning session. This limits the complexity of these algorithms, however allows adjustments to the scanner parameters to be made 'on the fly' to improve scan accuracy and also enables researchers to use real time fMRI (rtfMRI) experimental and therapeutic paradigms such as biofeedback therapy (e.g. Weiskopf et al., 2007). Offline motion correction algorithms, conversely, are applied after the entire scanning session is complete. Because they are not required to execute within one time of repetition cycle, offline motion correction algorithms are able to employ much more complex, compute-intensive optimization and interpolation algorithms and can prioritize accuracy as opposed to speed.

Another important distinction is between correcting for *physiological motion* versus *random motion*. Physiological motion refers to the cyclical motion of the head and blood vessels caused by respiration and pulse. Although these motions are relatively small, they can cause significant modulation of the BOLD signal (Noll & Schneider, 1994). Random motion refers to unintentional head movements caused by involuntary muscle twitches or an inability to maintain a stationary head position. The magnitude of these random movements is usually less than 1mm in normal subjects, but in certain special populations such as infants, the elderly, or the mentally ill these motions can be up to several millimeters (Friston et al., 1996).

It is also important to draw a distinction between *volume-by-volume* and *slice-by-slice* motion correction. In traditional fMRI scanning, each slice of each volume is acquired in series over time. As such, there is a choice in how many slices to treat as a unit when registering them. In volume-by-volume motion correction, the time difference between capturing each slice is ignored and all of the slices composing the subject's entire brain volume are treated as a whole. In this case, head movement is corrected for by aligning

each successive brain volume to a reference volume. The reference volume is usually the first volume of the scanning session; an average of several volumes can however also serve as this reference (Friston et al., 2006). Slice-by-slice motion correction, however, operates at a finer temporal granularity. In this case, each slice (or a collection of several slices, referred to as a *chunk*) is treated as a discrete unit and is aligned to a reference slice or reference chunk. Volume-by-volume motion correction has the benefit of faster processing time, because fewer alignments must be carried out; slice-by-slice motion correction is however able to compensate for greater magnitudes of motion which cause significant changes in brain position during the capture of a single volume (e.g. Speck, Hennig & Zaitsev, 2006).

In the case of volume-by-volume and slice-by-slice motion correction, the 3D imaging data being aligned are typically treated as a rigid body. Under the *rigid body assumption*, there are only six degrees of freedom (three rotational and three translational) along which an image can be transformed to align it with the template image. The rigid body assumption is generally valid because the brain and head move together during scanning; this assumption also serves to simplify the optimization step used in most motion correction algorithms.

Finally, the spatial transformation that a motion correction algorithm produces to align a given image back to the template image can be *linear* or *nonlinear*. Linear spatial transformations include translation, rotation and zooming. Most linear transformations preserve the rigid body assumption and do not deform the 3D brain image. Nonlinear spatial transformations include affine transformations and warps; these are most often used when registering a subject's scan data to a reference anatomical template to facilitate between-subject comparisons. Because nonlinear spatial transformations violate the rigid body assumption, most motion correction algorithms provide linear spatial transformation solutions.

Chapter 2

2 Interpolation Methods in fMRI Motion Correction

The interpolation method used in a given motion correction algorithm has a significant impact on the algorithm's overall performance (Jenkinson, Bannister, Brady & Smith, 2001). Interpolation is used in fMRI motion correction both to determine the values of voxels intermediate to the raw scan data during optimization of the spatial transformation (or motion estimate) and to produce the final scan session data once an accurate spatial transformation correcting for the subject motion in each volume has been determined.

The interpolation step of an iterative motion correction algorithm is also often its most compute intensive component. Correspondingly, the speed of the interpolation method will tend to dominate the running time performance of a motion correction algorithm; this is especially so if the algorithm needs to perform many iterations (and therefore interpolations) when determining an optimal spatial transformation (or motion estimate). In this chapter, a review of several methods for the 3D interpolation of fMRI neuroimaging data is provided; for each interpolation method, the structure of the algorithm and a prominent software package employing it are described.

2.1 Trilinear Interpolation

Trilinear interpolation is a multivariate interpolation method which allows for the interpolation of intermediate points on a regular 3D grid by chaining together multiple linear interpolations. Trilinear interpolation is one of the fastest 3D interpolation methods however it is often criticized for its potential inaccuracy.

2.1.1 Algorithm

Trilinear interpolation is algorithmically the simplest method of interpolation presented in this chapter. Given a set of known points on a regular 3D grid, trilinear interpolation uses a chain of 7 individual linear interpolations to approximate the value of any intermediate point contained within a rectangular prism given by the grid. Although trilinear interpolation is relatively simple conceptually and fast computationally, it has the

disadvantage of relative inaccuracy compared to other interpolation methods (Tong & Cox, 1999). Trilinear interpolation has also been criticized for introducing spatial smoothing when applied to fMRI data (Oakes et al., 2005).

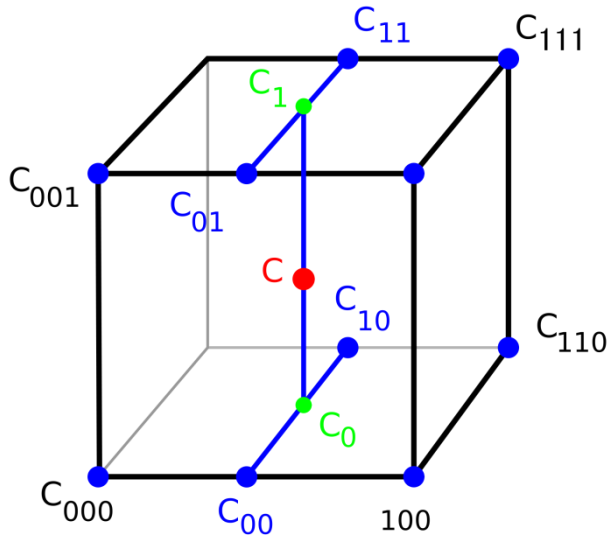

Figure 2: Trilinear interpolation (Wikipedia).

For a given intermediate point *c* whose value needs to be interpolated, the eight corners of a cube on the regular grid surrounding it are first found. Then, four intermediate points on the lines connecting those eight corners, referred to as c_{00} c_{01} c_{10} and c_{11}, are calculated using one dimensional linear interpolation. Next, two intermediate points on the lines connecting c_{00} c_{01} c_{10} and c_{11} are interpolated, referred to as c_1 and c_0. Finally, *c* is given by the linear interpolation of c_1 and c_0. These steps are shown in Figure 2.

2.1.2 AIR Software Package

In their 1998 paper, Woods et al. describe their Automated Image Registration (AIR) software package. This package contains an image registration method which functions very similarly to the general motion correction algorithm described in the previous chapter. This registration algorithm serves as the foundation for the AIR software package's motion correction utility, because as stated previously motion correction is actually a special case of image registration. In the AIR registration algorithm an original scan is first interpolated based on a possible solution spatial transformation, then a cost

function is evaluated which provides the algorithm with a quantitative measure of how well the images are registered, then finally an optimization function determines a new spatial transformation to apply to the image for the next cycle of the registration algorithm. A schematic of the AIR registration algorithm is shown in Figure 3.

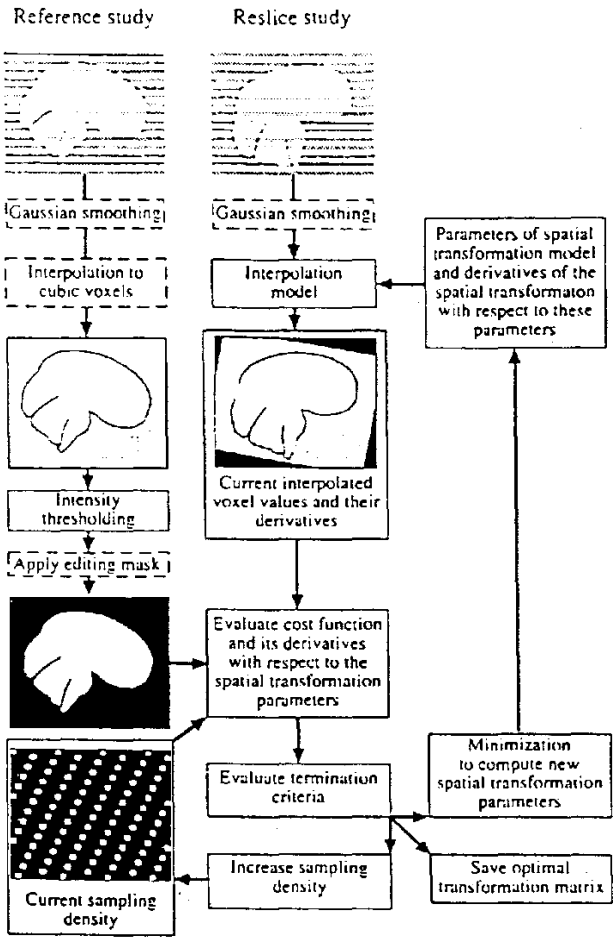

Figure 3: AIR registration algorithm (Woods et al., 1998).

To compare the images being registered, one of the images must be resampled according to the parameters of the current spatial transformation. This resampling requires that voxel intensities at locations in between the voxel locations represented in the original image be calculated. The AIR algorithm uses the trilinear interpolation method to perform this calculation. Once the final spatial transformation has been determined on the last iteration of the realignment algorithm, the AIR software package provides the option of using more advanced and accurate interpolation methods to produce the final image.

To determine the error for a given iteration of the realignment algorithm, the AIR software package uses the ratio image uniformity (RIU) cost function. To compute this cost function, a resampled image (given by a set of realignment parameters) is divided by the image to which it is being registered on a voxel-by-voxel basis to create a ratio image, with the uniformity of this image measured by its standard deviation. This standard deviation is then divided by the mean ratio to provide a normalized cost function value for the realignment parameters used to create the image. The minimization of this cost function therefore increases the uniformity of the ratio image independent of the global intensity scaling of the original images. The AIR software package also includes a second option for the cost function, namely a least squares approach similar to that used by Friston et al. (1996). The least squares cost function is given by the average voxel-by-voxel difference between the resampled image and the reference image. The AIR least squares cost function also adds an intensity scaling step to compensate for global discrepancies in image intensity.

To handle the iterative adjustment of the spatial transformation to find an optimal rigid body transformation of the brain image, the AIR software package uses a variation on the Powell optimization algorithm (Powell, 1964). This optimization is a conjugate direction method, searching through a 6D parameter space to find a local minimum of the error function used. Powell's method does not require that derivatives be taken (as does for example the Gauss-Newton method described below), but instead minimizes the error function using a bi-directional search along each vector in a set of search vectors, usually simply the normals of the search space aligned along each axis. As such, it is useful for calculating the local minimum of a continuous but complex non-differentiable function.

2.2 Spline Interpolation

Spline interpolation is a special case of polynomial interpolation, using a piecewise polynomial called a Basis spline. Splines were originally used to describe curves in shipbuilding and are now widely used in computer graphics. Spline interpolation is a special case of polynomial interpolation however it has several advantages over its more general progenitor.

2.2.1 Algorithm

Given a set of *n* unique one dimensional points, the 1D polynomial interpolation problem is to find the polynomial function which goes exactly through those points. The search for this polynomial is equivalent to solving a linear system of equations; with a polynomial of at least degree *n* - 1, there exists a provably unique solution to this linear system. The 1D case of polynomial interpolation is shown in Figure 4.

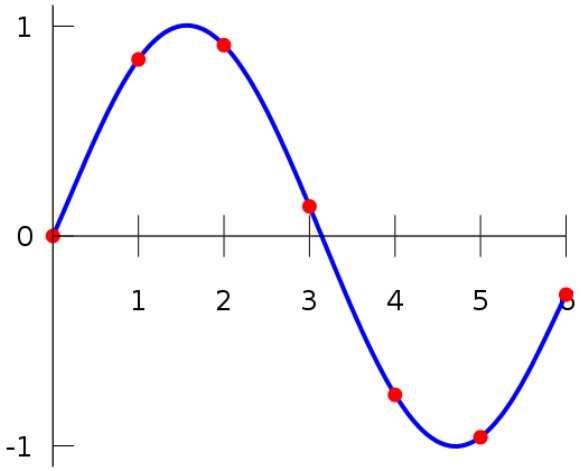

Figure 4: 1D Polynomial interpolation (Wikipedia).

Spline interpolation is a special case of polynomial interpolation using a Basis spline function. A Basis spline or B-spline is a piecewise polynomial function continuous at each piece boundary, called a knot. Given a set *s* of *n* unique points, spline interpolation will produce a piecewise polynomial function which passes through each knot point in *s* while minimizing the amount of bending within the function as a whole (Webster & Oliver, 2001). Typically, third degree polynomials are used for each piece of the function (idem); these are referred to as cubic splines. First and second degree polynomials can also be used however; these are referred to as linear and quadratic splines respectively. An example of 3D cubic spline interpolation from the GNU Octave software package is shown in Figure 5.

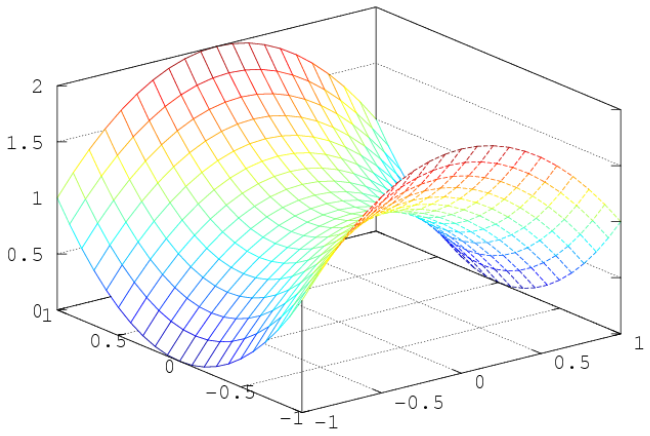

Figure 5: 3D Cubic spline interpolation (GNU Octave).

Spline interpolation is generally preferred over polynomial interpolation because of the low interpolation error which can be achieved even when using low degree polynomial functions for each piece of the function. Spline interpolation also avoids the problem of oscillation at the edges of the interval of interpolation, which can occur when fitting a high degree polynomial function to a set of equally spaced data points (also known as Runge's phenomenon; see Fornberg & Zuev, 2007).

2.2.2 SPM2 Software Package

Friston et al. present an "efficient, automatic, and general multidimensional nonlinear spatial transformation technique" (1995, p. 166). It was the authors' intention to create a general registration algorithm applicable for realigning variations and combinations of fMRI, structural MRI and positron emission tomography (PET) data. In order to accomplish this, the authors used two guiding principles in designing their algorithm.

First, the authors wanted the constraints on the image transformations their algorithm would use to be reasonable, explicit and operationally specified. To this end, the authors decomposed the differences between two images into two components: intensity differences between two images which are in perfect physical alignment and differences in physical alignment or size and shape of the object being scanned. As such, the image

transformations produced by their algorithm are a nonlinear combination of rotational, translational and intensity transformations.

The authors' second guiding principle was to develop a method with a single unique solution transformation for each volume to be aligned. To this end, the authors linearised the intensity transformation function using a low-order Taylor series approximation, ensuring that a single least squares solution exists for every image registration; a least squares solution uses an over-determined system of equations to minimize the sum of the squares of the errors (SSE) in the results of each equation. To minimize the sum of squared errors in the system of equations, the authors employed Gauss-Newton optimization. The Gauss-Newton optimization method iteratively finds the minimum SSE based on the first derivative of the sum of squared error function. This algorithm can produce general image registration solutions, however when used for fMRI motion correction the authors' algorithm uses only translations, rotations and an identity intensity transformation. That is, the solution transformation produced still follows the rigid body assumption.

This general nonlinear registration algorithm served as the foundation of the motion correction algorithm used in the neuroimaging data processing software package SPM2 (Statistical Parametric Mapping 2), maintained by the Wellcome Trust Centre at University College London. SPM2 uses a registration algorithm closely based on that presented in Friston et al.'s 1995 paper to provide estimates of subject motion, then using those estimates determines a solution spatial transformation to correct for that motion. Once a solution transformation has been determined, the original scan data is interpolated using a 4D Basis spline to produce the final motion corrected scan data.

2.3 Fourier Interpolation

Fourier interpolation is lauded for combining speed and accuracy when interpolating fMRI data. Because it operates on data in the Fourier domain, the native k-space of raw scanner data, Fourier interpolation has seen wide use in medical imaging in general.

2.3.1 Algorithm

Consider a spatial transformation which will rotate a slice from an fMRI volume. To determine the new voxel intensities for the slice given that spatial transformation, voxel intensity values intermediate to the existing image must be computed. Fourier interpolation computes these intermediate voxel intensities by first converting the spatial transformation from an image space transformation to a k-space transformation. This new transformation is then applied to the raw k-space data and the result is used to produce the new voxel intensities though the application of an inverse Fourier transform:

$$F(x) = \int_{-\infty}^{\infty} F(k) e^{2\pi i k x} \, dk$$

2.3.2 AFNI Software Package

Cox and Jesmanowicz (1999) describe a fast and accurate method for shifting and rotating a 3D image using a shear factorization of the rotation matrix, as is used to handle motion correction in the AFNI (Analysis of Functional Neuroimages) software package (Cox, 1996). The authors based their method on the work of Eddy, Fitzgerald and Noll (1996) who proposed the combination of three 2D shearing operations and Fourier transform based shifting for 2D MRI rotation. The authors extended this previous work, relying on the principle that a 3D proper orthogonal matrix can be factored into three 2D rotations and so a general 3D image rotation can be accomplished with nine 2D shears. As such, a 3D shear factorization has the same advantage that a 2D shear does in that its elementary operations are coordinate shifts on 1D rows extracted from the image. In the authors' algorithm, when any particular row of the 3D image is shifted in this way, the row's data are interpolated using fast Fourier transforms (FFTs).

To determine the correct rotation in order to register a given volume to a template image, the authors repeatedly linearised a weighted least squares penalty (or error) function with respect to a rigid body transformation of the brain. This error function is:

$$E(a) = \sum_{x} w(x)[J(T[a])x - I(x)]^2$$

Repeated linearization is equivalent to applying an iterative gradient descent algorithm to the least squares penalty function. Gradient descent works by stepping toward the next lowest point in the error space of the least squares penalty function at each iteration of the optimization algorithm. The size of the step taken at each iteration is based on the first derivative of the error function on that iteration. Gradient descent can work in an error space of any number of dimensions and is guaranteed to converge to a local minimum, however convergence can be slow close to minima because the first derivatives of the error function surrounding them are typically small. When the error function has been minimized, the corresponding 3D shear image rotation will align the current image to the template image and thereby correct for subject motion.

2.3.3 PACE Motion Correction

The PACE (Prospective Acquisition CorrEction) motion correction method presented by Thesen, Heid, Mueller and Schad (2000) departs from retrospective motion correction techniques, where motion is corrected for by processing the data after a full set of scan data have been acquired. PACE instead corrects for motion and updates the scanner parameters for slice orientation and position in real time after each volume is captured (the principle flow chart of the complete PACE real-time acquisition correction is shown in Figure 6). After each volume is acquired, the motion of that volume relative to a reference volume is detected, then those positional data are sent simultaneously to the scanner and to a Fourier interpolation algorithm (the authors refer to interpolation as regridding).

To detect the motion in each volume during the scan, PACE uses as a similar technique to Friston et al. (1996). First, one volume is chosen as a reference to which all subsequent volumes will be aligned. To speed up computation, the PACE algorithm uses only a subset of voxels in the brain scan to perform the alignment, covering roughly the area containing brain tissue in the interior slices of the volume. Next, the rigid body transformation mapping the current volume being aligned to the reference volume is expanded as a first order Taylor series. This Taylor series is approximated and a least squares solution for its parameter function is obtained iteratively (with the motion parameters or spatial transformation resampled at each iteration). When complete, this

algorithm produces a rotation matrix and a shift vector describing the motion between the most recently acquired volume and the reference volume. The authors suggest that for typical fMRI scans, the motion in any given volume can be detected with only ten iterations of this function.

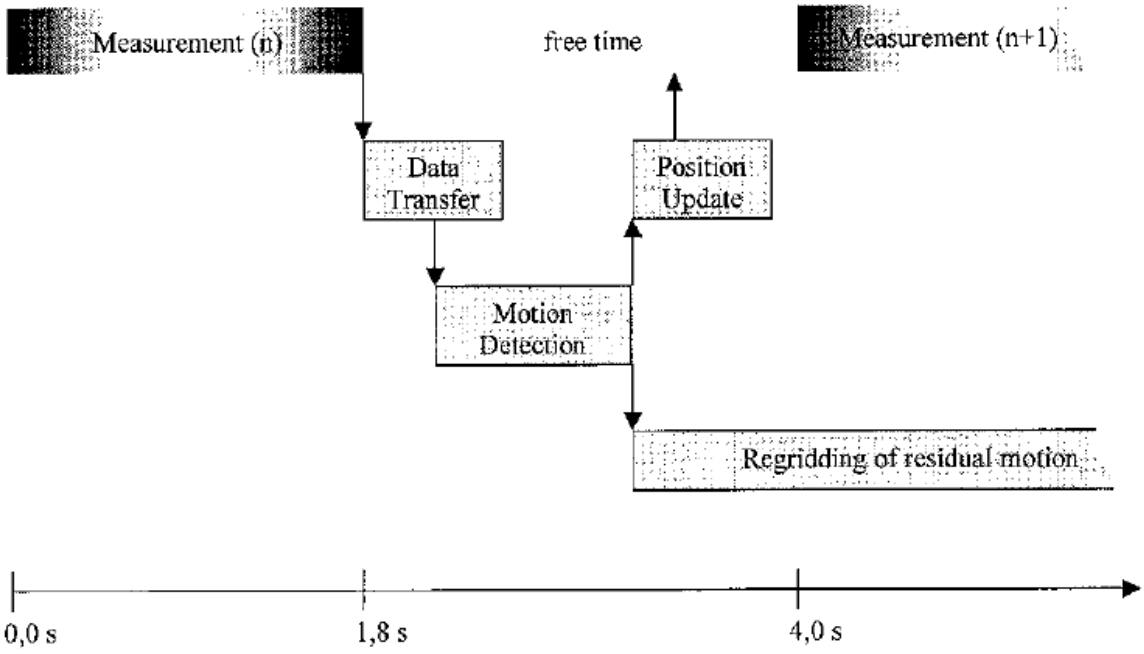

Figure 6: PACE flowchart (Thesen, Heid, Mueller & Schad, 2000).

This method of prospective motion correction feeds back the positional information calculated from a given volume to the next volume in the scanning session, however because of the 3-4 second delay in volume acquisition for the scanner the authors were using, their algorithm also includes a step to remove any residual motion not corrected for by the use of updated positional parameters during image acquisition. For each acquired volume, the transformation required to adjust the current image to the reference image are calculated, then this transformation is used to regrid (interpolate) the measured volume in order to eliminate residual volume to volume motion from the final session data. The authors chose to use Fourier interpolation, specifically the shearing method introduced by Eddy, Fitzgerald and Noll (1996) described above to accomplish this interpolation.

2.4 Sinc Interpolation

Like spline interpolation, sinc interpolation is a special case of the more general polynomial interpolation. Sinc interpolation has the unique advantage of being able to provide an almost perfect reconstruction of band-limited data. Although it provides high quality results, sinc interpolation is very slow relative to other interpolation algorithms (Friston et al., 1996).

2.4.1 Algorithm

A detailed exploration of the 3D sinc interpolation method is provided in Chapter 3.

2.4.2 FSL MCFLIRT Software Package

Jenkinson, Bannister, Brady and Smith (2001) wanted to address the problem of the optimization method used in motion correction algorithms; a problem they argue had received little attention at the time. Specifically, the authors argue that most optimization algorithms in use at the time of their writing were susceptible to becoming trapped in local minima of the so-called error space of the optimization function. That is, the optimization algorithm might become caught in a ‘large scale basin’ or a ‘smalls scale dip’ and fail to reach a global minimum for the cost function.

In the former case, an optimization algorithm would produce a large misregistration because the local minimum of a large scale basin is far from the global minimum. In the latter case, the optimization algorithm simply stops prematurely at a small scale dip, causing a large misregistration at low resolutions or a small registration at high resolutions. To combat these two types of local minimum optimization errors, the authors used a two-pronged approach; apodization of the cost function (that is, smoothing the function at its edges) to eliminate the ‘small dip’ error, combined with a hybrid global-local optimization technique which utilizes prior knowledge about the transformation parameters and typical data size to avoid ‘large scale basin’ errors.

The mathematical details of how the authors apodized the cost function are outside the scope of this paper, however their global-local hybrid optimization method warrants further explanation. This method is designed to provide a reliable estimate of the global

minimum of the cost function given some time restriction. The method uses four stages of search, each with the template and image to be aligned scaled to a different resolution (see Figure 7). At each stage, whenever the global-local hybrid optimization method is minimizing the cost function, Powell minimization is used.

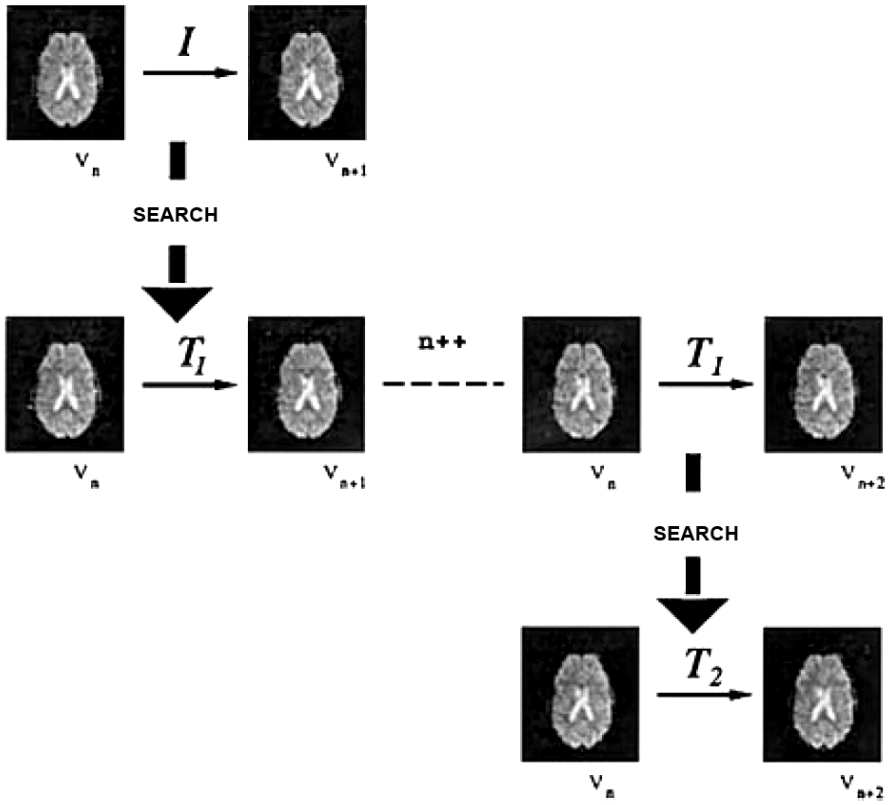

Figure 7: FSL global-local optimization
(Jenkinson, Bannister, Brady & Smith, 2001).

In the first stage, the images are pre-blurred with a Gaussian kernel and voxels are scaled to 8 x 8 x 8mm, preserving only the gross image features. The search for a minimum of the cost function at the 8mm stage is divided into three steps: step one, a coarse search over the rotation parameters with a full local optimization of translation and scale for each rotation tried; two, a finer search over rotation parameters but with only a single cost function evaluation at each rotation; three, a full local optimization (rotation, translation and global scale) for each local minimum detected from the previous stage (ibid., p. 831). Although it is unlikely that the first step will provide an accurate estimation of brain

motion, the progression of these steps provides several reasonable potential starting points in the error function for the next stage of the optimization.

The second stage of the optimization is performed at 4 x 4 x 4mm scale (images are again pre-blurred with a Gaussian kernel). This stage takes the top three local minima candidates for the global minimum, then also generates six rotational and four scale perturbations of each of these minima, producing 33 candidate starting points for a multi-start search of the error function space. After each of these candidate motion estimations have been minimized for error, the single best candidate is selected for further optimization in the next stage of the method.

The third stage of the algorithm works with the images scaled to 2 x 2 x 2mm, where skews and anisotropic scalings begin to become significant. Consequently, the authors' method progressively introduces these extra degrees of freedom (DOF) to the error minimization function by calling the local optimization method three times: first using only 7 DOF (rigid body and global scale), then with 9 DOF (rigid body and independent scalings), then with the full 12 DOF (rigid body with scales and skews). After the optimization has run with these additional degrees of freedom, the current motion estimate is passed to the next stage of the method.

In the last stage of global-local hybrid optimization method, with the image scaled to 1 x 1 x 1mm, the cost function evaluations take 8 times longer than at the 2 x 2 x 2mm scale and 512 times longer than at the 8 x 8 x 8mm scale. As such, during this stage only one pass of the Powell local optimization algorithm is performed, the result of which is the final registration solution.

When the global-local hybrid optimization method is applied to the problem of motion correction, as in the FSL (fMRIB Software Library) software package's MCFLIRT (Motion Correction fMRIB Linear Image Registration Tool) utility, the middle image of the scan series is taken as the template to which all other volumes are registered. Furthermore, the final two stages of optimization at the 2mm and 1mm scale are omitted. When the image registration transformation is computed, sinc interpolation is used to interpolate the final corrected data.

2.5 Other Motion Correction Techniques

In this chapter several motion correction algorithms from the past 20 years have been discussed. Most of these algorithms rely on the iterative adjustment of an estimate of subject motion using an optimization algorithm alongside a cost function describing the error difference between an uncorrupted image and a template image (e.g. Jenkinson, Bannister, Brady & Smith, 2001; Thesen, Heid, Mueller & Schad, 2000; Woods et al., 1998). Each of the algorithms relying on this optimization process to perform motion correction varies in its computation of the cost function, the optimization algorithm used and the nature of the estimation of the motion. Other approaches, such as predictive approaches using a priori estimates of physiological motion are also possible (e.g. Glover, Li & Ress, 2000; Hu, Le, Parrish & Erhard, 1995) and instead rely on removing trends in the raw data that are correlated with known periods of subject motion.

A class of motion correction approaches which have not been reviewed here are those which use external markers to monitor the movement of the subject's head during scanning to provide the estimates for motion, then remove that estimated motion from the scan data. This can be done either retrospectively after data acquisition is complete (e.g. Tremblay, Tam & Graham, 2005) or prospectively while data is being acquired (e.g. Zaitsev et al., 2006). Furthermore, the modality of this positional monitoring can vary, from optical markers to track motion (as in the previous two examples) or radio frequency markers (Ooi et al., 2009).

Chapter 3

3 Three Dimensional Sinc Interpolation

As demonstrated in the previous chapter, several different 3D interpolation methods are currently in use for fMRI motion correction. Of these methods, sinc interpolation was selected to demonstrate the importance of the programming language and the hardware platform used for executing a neuroimaging data processing algorithm.

Sinc interpolation was chosen for this demonstration for two reasons. It is considered among the best interpolation methods for 3D data interpolation (Tong & Cox, 1999) and as such the development of new and faster implementations will service the neuroscientific community. Secondly, other methods of 3D data interpolation such as trilinear interpolation and spline interpolation have sufficiently short running times that the differences between different implementations could be insubstantial; sinc interpolation however has a long running time (Friston et al., 1996) and therefore would best serve to demonstrate inter-implementation differences.

3.1 Algorithm

Sinc interpolation, also known as the Whittaker-Shannon interpolation method, is based on the unnormalized sinc function. The term sinc is a contraction of the Latin sinus cardinalis meaning cardinal sine, with the unnormalized sinc function is defined as

$$sinc(x) = \frac{sin(x)}{(x)}$$

for $x \neq 0$, with sinc(0) = 1. Sinc interpolation is commonly employed in digital signal processing for the band-limited interpolation of discrete-time signals. Band-limiting is the limiting of a signal's Fourier transform to zero above a certain finite frequency. Band-limiting is an important concept within the context of sinc interpolation, because a band-limited signal can be fully reconstructed from its samples, provided that the sampling frequency exceeds twice the maximum frequency in the band-limited signal; the minimum sampling rate providing a full reconstruction of a band-limited signal is

referred to as the Nyquist frequency. In essence, with a band-limited signal sinc interpolation can be used to correctly compute signal values at arbitrary continuous times from a discrete set of samples provided they are sampled at a rate above the Nyquist frequency. The standard sinc interpolation formula is

$$x(t) = \sum_{n=-\infty}^{\infty} x_n \times sinc\left(\frac{\pi}{T}(t - nT)\right)$$

which can be expressed using only the sine function as

$$x(t) = \sum_{n=-\infty}^{\infty} x_n \times \frac{sin\left(\frac{\pi}{T}(t - nT)\right)}{\left(\frac{\pi}{T}(t - nT)\right)}$$

where T is the sampling period used to determine x_n and $x(t)$ is the reconstructed signal. The above formula represents a linear convolution between the sequence and scaled and shifted samples of the function (Oppenheim & Schafer, 1975).

Hajnal et al. (1995) describe an expansion of the standard sinc interpolation formula allowing for the 3D interpolation of neuroimaging data. This expansion is accomplished with a cosine Hann (Hanning) window using the normalized sinc function

$$sinc(x) = \frac{sin(\pi x)}{(\pi x)}$$

In this formulation, the intensity value for a given voxel is the multiplicative combination of three 1D sinc interpolations, with one interpolation for each dimension. The intensity value I for voxel at (x, y, z) is defined using a Hann sinc interpolation as

$$I(x, y, z) = \sum_{X}\sum_{Y}\sum_{Z} I(X, Y, Z) \times HS(x, X, R) \times HS(y, Y, R) \times HS(z, Z, R)$$

with $HS(a,A,R)$ defined as

$$HS(a, A, R) = \frac{sin\big(\pi(a - A)\big)}{\pi(a - A)} \times \frac{1}{2}\left[1 + cos\left(\frac{\pi(a - A)}{\mathrm{R} + 1}\right)\right]$$

where *X*,*Y*,*Z* are the coordinates of the original data set and R is the size of the Hann window used. This Hann function eliminates problems with oscillatory effects at discontinuities in the function and guarantees that the convolution coefficients fall off to zero at the edge of the Hann window (Thacker, Jackson, Moriarty & Vokurka, 1999). A graph of the Hann window function as applied to audio samples is show in Figure 8 (Wikipedia). Throughout this document the collection of voxels in a given volume covered by the Hann window is referred to as a *sinc kernel* and to the portion of the final interpolated intensity *I* given by one of the voxels in a sinc kernel as that voxel's *sinc contribution*. Throughout the remainder of this document, the size of a sinc kernel is referred to by its radius; that is, a sinc kernel of size 7 x 7 x 7 will have 7 voxels between the intermediate voxel *v* and an outer plane of the 3D kernel. Figure 8 below shows an illustration of a 4 x 4 x 4 sinc kernel.

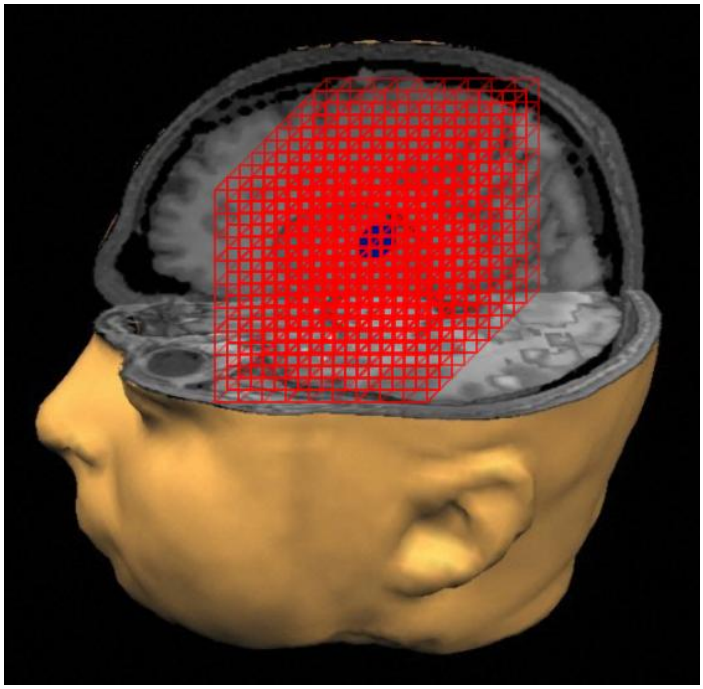

Figure 8: Illustration of a 4 x 4 x 4 sinc kernel.

The asymptotic time complexity of the serial sinc interpolation function depends on whether the number of voxels to be interpolated or the size of the kernel is varied. In the case that the number of voxels is varied, the sinc interpolation function has a complexity of O(n). If the size of the sinc kernel is varied however, the complexity will be $O(n^3)$.

3.2 Serial Implementation

As explained in Chapter 1, during motion correction new coordinates for each voxel in an fMRI volume are given by the solution spatial transformation computed by the optimization algorithm. Because these coordinates lie between the existing coordinate grid of the raw scan data, an estimate of the intensity value for each of these intermediate voxels must be calculated. In this section the structure and function of a serial algorithm for the 3D sinc interpolation of the intensity at an arbitrary point within a fMRI brain scan volume is presented, which for convenience is referred to as *serialSinc*. serialSinc is based on the AIR5 3D sinc interpolation algorithm (Woods, Cherry & Mazziotta, 1992).

The serialSinc algorithm takes as input an array of coordinates, intermediate to the existing grid structure of the raw fMRI data, whose intensity values need to be interpolated; the input array of intermediate coordinates is referred to in serialSinc as a *chunk*. The number of voxels in a chunk can vary depending on whether the motion correction algorithm in question operates slice-by-slice, volume-by-volume, or somewhere in between. As such, serialSinc is designed to handle an arbitrary number of voxels per chunk. The number of voxels in the chunk is referred to as the *chunk size*.

serialSinc uses the Hann windowed 3D sinc interpolation method described above to compute the new intensity value for each intermediate voxel coordinate. As explained in the previous section, the new intensity value for a given voxel *v* is the sum of the sinc contribution of each known voxel in its surrounding sinc kernel. The sinc contribution for a given voxel in the sinc kernel is computed by multiplying its one dimensional sinc contribution (given by the Hann windowed normalized sinc function) in each of the x, y and z directions by its intensity. In serialSinc, the sinc contribution of each voxel in the sinc kernel is computed serially and added to the total for the interpolated intensity of voxel *v*. This total is then stored in an array holding the new interpolated intensities for every voxel in the chunk. The size of the Hann window and therefore the size of the sinc kernel along the x, y and z dimensions is passed into serialSinc as a parameter.

The serialSinc algorithm is shown in pseudocode below, alongside the sinc function, which in practice would simply replace the call to sincFunction in serialSinc.

```
// sincFunction

GET voxelPosition
GET kernelCenter
GET kernelSize

IF voxelPosition == kernelCenter
    SET result as 1.0

ELSE
    SET result as sin(pi*voxelPosition)/(pi*voxelPosition) * \
                          0.5*(1.0 + cos((pi*voxelPosition)/kernelSize))

RETURN result

// serialSinc Algorithm

GET kernelSizeX
GET kernelSizeY
GET kernelSizeZ

GET chunkXSize
GET chunkYSize
GET chunkZSize
GET chunkCoordinates

INIT interpolatedChunkIntensities

FOR kernelCenterZ in chunkZSize
    FOR kernelCenterY in chunkYSize
        FOR kernelCenterX in chunkXSize

            COMPUTE kernelBoundsX from kernelSizeX and kernelCenterX
            COMPUTE kernelBoundsY from kernelSizeY and kernelCenterY
            COMPUTE kernelBoundsZ from kernelSizeZ and kernelCenterZ

            SET kernelTotal to 0

            FOR currentVoxelPositionZ in kernelBoundsZ

                COMPUTE sincz = sincFunction(currentVoxelPositionZ)

                FOR currentVoxelPositionY in kernelBoundsY

                    COMPUTE sincy = sincFunction(currentVoxelPositionY)
                    SET sinczy to (sincy * sincz)

                    FOR currentVoxelPositionX in kernelBoundsX

                        COMPUTE sincx =  sincFunction(currentVoxelPositionX)
                        SET sinczyx to (sincx * sinczy)

                        GET currentVoxelIntensity
                        SET newVoxelIntensity to (currentVoxelIntensity * sinczyx)
                        ADD newVoxelIntensity to kernelTotal

            STORE kernelTotal in interpolatedChunkIntensities
```

3.3 Parallel Implementation

Parallel computing in general is based on the principle that large computing problems can be divided into many smaller problems, all of which can be completed simultaneously. The two primary paradigms in parallel computing are *data parallelism* and *task parallelism*. Task parallelism achieves improvements in the execution time of an algorithm by running sections of it concurrently. A section of code which is designed to run in parallel with other copies of itself is called a *kernel* (referred to herein as a *parallel kernel* to avoid confusion with sinc interpolation kernels). Each copy of the kernel is executed in its own *thread*; a serial algorithm will run in only one thread executing a repeated section of code thousands of time in sequence, whereas a parallel algorithm could run thousands of copies of a kernel in hundreds of threads simultaneously, depending on the hardware architecture employed.

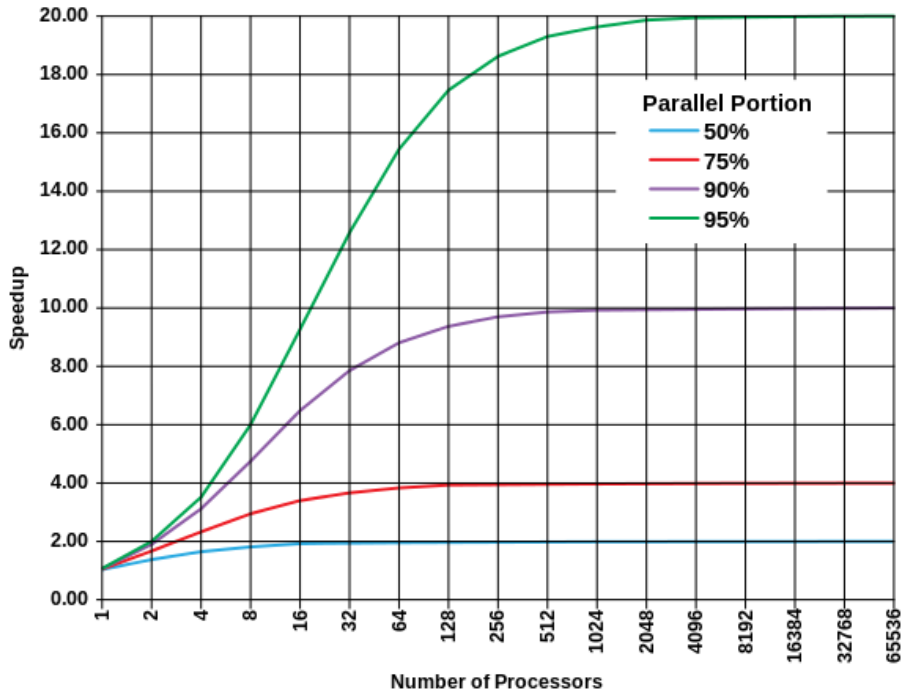

Figure 9: Amdahl's Law for 50%, 75%, 90% and 95% parallelized algorithms.

The process of converting a purely serial program to a logically equivalent parallel implementation is referred to as *parallelization*. Once parallelized, the remainder of the original program which executes in serial (and handles calls to execute the kernel) is

called the *host code*. Whether a given section of code in an algorithm can be parallelized depends on the logical structure of the algorithm; only subsections of an algorithm which are logically independent, meaning that they do not rely on each other's results and could be executed in any order, can be extracted into a parallel kernel and executed in parallel. Algorithms vary widely in the degree to which their logical structure can be executed in parallel, which affects the maximum performance benefits which can be achieved by parallelizing them. The relationship between the maximum expected improvement to an algorithm's overall running time for a given number of processors when some percentage of the algorithm is parallelized is referred to as *Amdahl's Law* (Amdahl, 1967). Figure 9 above shows Amdahl's Law for algorithms with parallel portions of 50%, 75%, 90% and 95%.

Although significant speedups can be achieved by parallelizing an algorithm and running many copies of a kernel in multiple concurrent threads, there is an overhead introduced by running multiple threads referred to as the *burden of parallelism*. If each thread does not contain a sufficient computational load when the program is parallelized, then the burden of parallelism will outweigh the gains afforded by executing that code concurrently. The size of the burden of parallelism and how my threads can execute concurrently both depend on the hardware architecture used. How much computation there is in each thread compared to the communication between a thread and the host code is referred to as the *granularity* of the algorithm. Fine (or high) granularity refers to a smaller ratio between computation and communication; decreasing the granularity of a parallelized algorithm by increasing the computational load in one thread will typically decrease the burden of parallelism.

The serialSinc algorithm presented in the previous section has a very high degree of task parallelism. Because the sinc kernel uses only the intensities of the raw data which do not change, the intensity for each intermediate voxel in a chunk can be computed in parallel with every other voxel in that chunk. Going further, within the calculation of an intermediate voxel's new interpolated intensity, the sinc contribution of each voxel in the sinc kernel can be computed in parallel; the sinc contribution of each voxel within a sinc kernel is independent and is simply added to the cumulative total for intensity of the

voxel at the center of the kernel. Finally, the determination of the 1D sinc function coefficient used to compute each voxel's sinc contribution can also be completed in parallel.

The parallelized version of the serialSinc algorithm from the previous section is called parallelSinc. The parallelSinc algorithm computes the interpolated intensity for each intermediate voxel in the chunk in parallel; the calculation of each voxel's intensity however is still performed in serial. This design decision was made to ensure there would be enough computation performed in each thread, such that the burden of parallelism would not outweigh the benefits of parallelization.

parallelSinc uses the same pseudocode to replace calls to sincFunction as shown above. The parallelSinc algorithm is shown in pseudocode below.

```
// parellelSinc Algorithm

GET kernelSizeX
GET kernelSizeY
GET kernelSizeZ

GET kernelCenterX
GET kernelCenterY
GET kernelCenterZ

COMPUTE kernelBoundsX from kernelSizeX and kernelCenterX
COMPUTE kernelBoundsY from kernelSizeY and kernelCenterY
COMPUTE kernelBoundsZ from kernelSizeZ and kernelCenterZ

SET kernelTotal to 0

FOR currentVoxelPositionZ in kernelBoundsZ

    COMPUTE sincz = sincFunction(currentVoxelPositionZ)

    FOR currentVoxelPositionY in kernelBoundsY

        COMPUTE sincy = sincFunction(currentVoxelPositionY)
        SET sinczy to (sincy * sincz)

        FOR currentVoxelPositionX in kernelBoundsX

            COMPUTE sincx = sincFunction(currentVoxelPositionX)
            SET sinczyx to (sincx * sinczy)

            GET currentVoxelIntensity
            SET newVoxelIntensity to (currentVoxelIntensity * sinczyx)
            ADD newVoxelIntensity to kernelTotal

STORE kernelTotal in global memory
```

Chapter 4

4 Performance Benchmarking

To demonstrate the effects of the programming language and hardware platform used to implement the 3D sinc interpolation algorithm described in Chapter 3, the results of extensive benchmarking and performance testing are presented in this chapter. The data in this chapter are all presented graphically, however tables containing their exact values are provided in the appendices.

4.1 Experimental Setup

Consider the position of a neuroscience researcher who is trying to decide whether an algorithm will be suitable to include in her neuroimaging data analysis pipeline. In order to assess the viability of a given algorithm, this chapter demonstrates that the researcher must consider carefully both the programming language (and version thereof) used to implement it and the computer system or hardware platform used to run it. Each such computer system hardware platform is referred to as a *test bed*.

For each of three different programming languages, the differences which can be seen in the running time performance of the same algorithm across multiple test beds and hardware platforms (the test beds are outlined in Section 4.2 below) are investigated. This investigation serves to underscore the idea that the differences between hardware architectures which are contemporary to one another can have a significant impact on an algorithm's running time performance even within one language.

To conduct this investigation, the 3D sinc interpolation algorithms described in Chapter 3 were implemented in Python, C and OpenCL (Open Computing Language). A Python host program was used to handle file I/O of the test neuroimaging data and benchmarking the performance of the algorithm for each language. The python implementation of 3D sinc interpolation ran natively within this host, while the code for the C implementation was extracted into its own Python extension module using Python's distutils library (default compiler flags). The OpenCL code was directly embedded in the Python host

script using PyOpenCL. A list of software packages and compilers used and the versions thereof is provided in Appendix A.

For each programming language, the test bed hardware platform was compared over sinc kernel size because the size of the Hann window used for 3D sinc interpolation has a substantial effect on the accuracy of the interpolation. The so-called gold standard for 3D MRI data interpolation is a sinc kernel of size 13 x 13 x 13 (Jenkinson, Bannister, Brady & Smith, 2001), however sinc interpolation kernels of this size are very computationally expensive; therefore 4 different sized Hann windows (1, 3, 7, 13) were used to assess the performance of less accurate, however less expensive, sinc kernels. Throughout each of these comparisons, the number of interpolations was kept constant; 230400 interpolations were performed, corresponding to interpolating the voxel intensities on a regular grid intermediate to a raw data set for an entire fMRI brain volume of 80 x 80 x 36 voxels. The raw data for interpolations using smaller input (chunk) sizes are found in the appendices.

4.2 Test Beds

The first test bed is a powerful desktop workstation, described in Table 1. The second test bed is a GPGPU (general purpose graphics processing unit) laboratory desktop computer, shown in Table 2. The third test bed is a rack-mount server, shown in Table 3. The final test bed is an older laptop computer, included to show the relative performance of a legacy machine. The specifications for this test bed are shown in Table 4.

Table 1: Workstation desktop test bed specifications.

Operating System	Windows 7 Professional 64bit 6.1
CPU Model	AMD Phenom II Thuban 1090T Black Edition
CPU Clock	4026Mhz
CPU Cores / Threads	6 / 6
GPU Chipset	Radeon HD 6850
GPU Core Clock	870Mhz
GPU Architecture	960 Stream Processors
GPU Memory	1GB GDDR5 @ 1050Mhz

Table 2: Laboratory desktop test bed specifications.

Operating System	Ubuntu 64bit 3.8.0
CPU Model	Intel Xeon E5504, Intel Xeon E5504
CPU Clock	1596Mhz
CPU Cores / Threads	8 / 8
GPU Chipset	nVIDIA Tesla c2070
GPU Core Clock	1150Mhz
GPU Architecture	448 Stream Processors
GPU Memory	6GB GDDR5 @ 1500Mhz

Table 3: Rack-mount server test bed specifications.

Operating System	GNU Linux 64bit 2.6.32
CPU Model	Intel Xeon E5603
CPU Clock	1197Mhz
CPU Cores / Threads	4 / 4

Table 4: Legacy laptop test bed specifications.

Operating System	Windows 7 Professional 64bit 6.1
CPU Model	AMD Turion x2 TL-56
CPU Clock	1800Mhz
CPU Cores / Threads	2 / 2

4.3 Python

The results of comparing the performance of a Python implementation of the serialSinc algorithm across the four test bed hardware platforms described above are shown in Figure 10. The raw data for this comparison are provided in Appendix C. Each running time presented is the mean of 5 trials of executing the algorithm. Because of the three nested FOR loops used to iterate through the voxels in a sinc kernel, as the size of sinc kernel along the three dimensions increases the running time increases as a cubic polynomial with a constant input size.

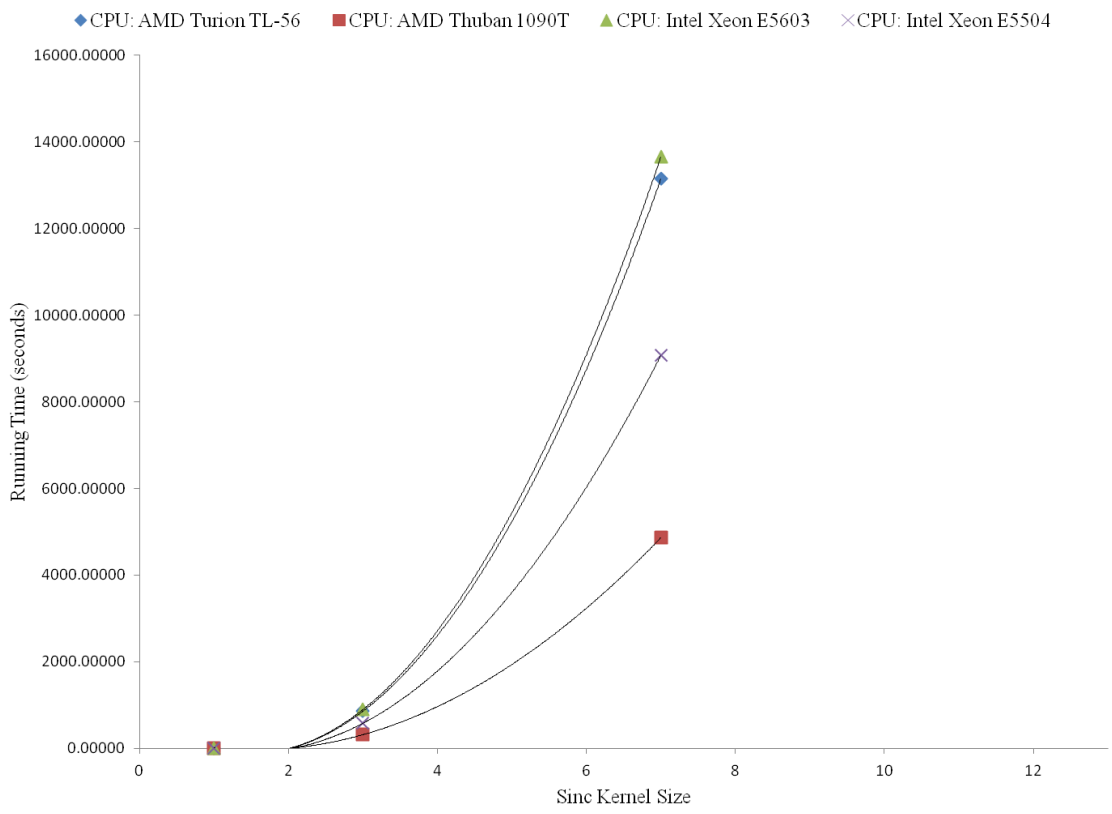

Figure 10: Python implementation running time comparison across test beds.

This performance comparison shows several interesting differences between the four test beds when using a sinc kernel of size 7 x 7 x 7. Firstly, one might expect that the two Intel processors, being of similar model numbers and specifications, would have similar performance with the legacy laptop lagging behind. Instead, the AMD Turion TL-56 and the Intel E5603 perform quite similarly at 13147 seconds and 13647 seconds respectively, however still with a significant difference in performance between them ($F(2,5) = 15700$, $p < 0.000001$). These two processors both execute the algorithm much more slowly than the Intel Xeon E5504 at 9069 seconds ($F(2,5) = 6888131$, $p < 0.000001$). Also notice that the AMD Turion TL-56, despite its faster clock speed, does not outperform the Intel Xeon E5504. This gap in performance could be explained by AMD Turion TL-56's small cache compared to the Intel Xeon E5504: 1024KB L2 cache versus 4096KB L2 cache respectively. The AMD Thuban 1090T, with a high clock speed and advanced architecture, outperforms the next fastest processor with a running time of 4862 seconds ($F(2,5) = 1015011$, $p < 0.000001$).

Importantly, regardless of the test bed hardware platform, the Python implementation of the 3D sinc interpolation algorithm was not able to interpolate the voxel intensities for an entire fMRI time series with a sinc kernel size of 13 x 13 x 13 within a tractable length of time. Extrapolating based on the performance of the fastest hardware platform, the AMD Thuban 1090T, it would take an estimated 21195 seconds or 5.9 hours to complete the interpolation of one volume; extrapolating based on the performance of the slowest hardware platform, the Intel Xeon 5603, it would take an estimated 60577 seconds or 16.8 hours to interpolate one volume. This means that at best, a Python implementation of the serialSinc algorithm could interpolate the data from an example 800 volume scan (20 minutes, TR = 1.5 seconds) in slightly over 6 months; surely an intractable length of time.

4.4 C

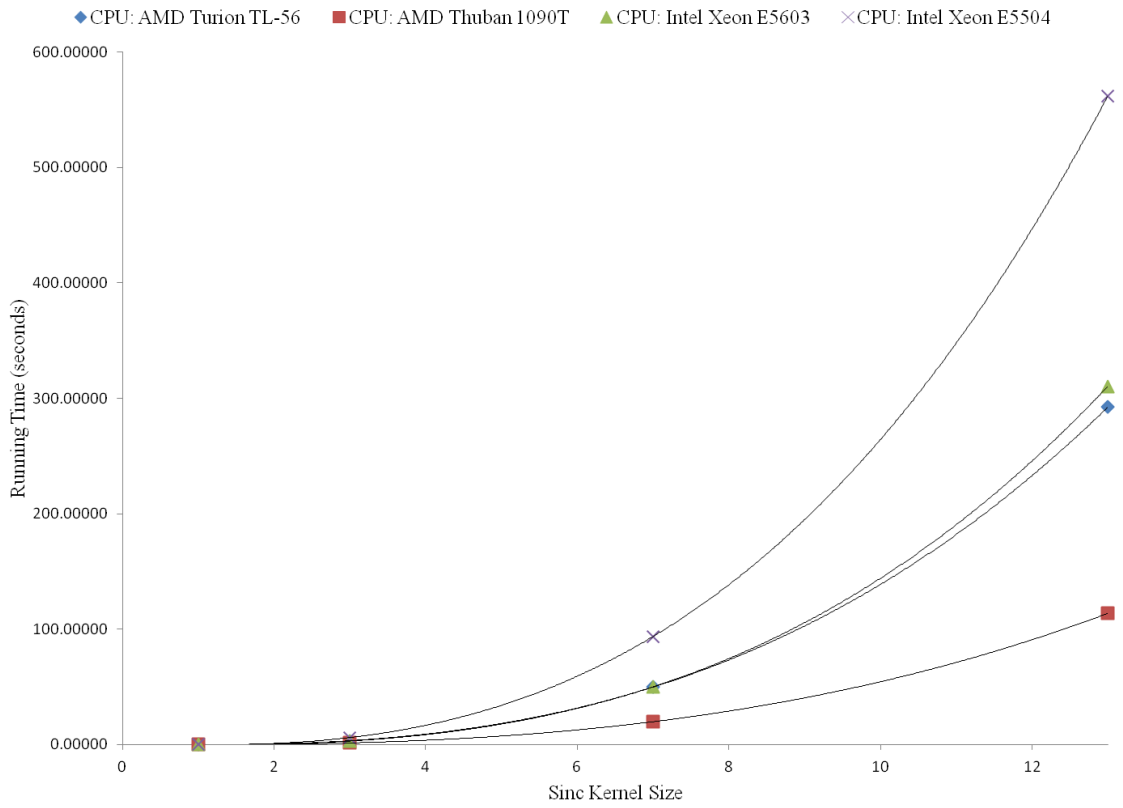

Figure 11: C implementation running time comparison across test beds.

The results of comparing the performance of a C implementation of the serialSinc algorithm across the four test bed hardware platforms described above are presented in Figure 11 above. The raw data for this comparison are provided in Appendix D. Each running time presented is the average of 10 trials of executing the algorithm. Once again, for all four test beds the running time increases as a cubic polynomial.

These results show several interesting similarities and differences as compared to the results of the Python implementation. Once again, the AMD Thuban 1090T outperforms all of the other processors with a running time of 114 seconds for a sinc kernel of size 13 x 13 x 13 when interpolating the voxel intensity values for a whole brain image ($F(2,10) = 1032026$, $p < 0.000001$). The AMD Turion TL-56 and the Intel E5603 again perform quite similarly at 292 and 310 seconds respectively, however still with a significant difference between them ($F(2,10) = 376167$, $p < 0.000001$); this implies that the similarities between these hardware platforms have a consistent impact on running time across different programming languages. Differing from the results of the Python implementation, the Intel Xeon E5504 performs substantially worse than the AMD Turion TL-56 and the Intel E5603, with a running time of 561 seconds ($F(2,10) = 104313253$, $p < 0.000001$).

The C implementation of serialSinc once again does not provide a tractable solution for performing 3D sinc interpolation with a kernel of size 13 x 13 x 13 for an entire fMRI time series, regardless of the hardware platform. When using the slowest test bed, the Intel Xeon E5504, interpolating an entire volume takes 561 seconds. It would therefore take approximately 125 hours or 5.2 days to interpolate an entire 800 volume time series. Using the fastest hardware platform, the AMD Thuban 1090T, it takes approximately 114 seconds to complete the interpolation of one volume, translating to approximately 25 hours for the interpolation of an entire time series. Although this computation time is much faster than that of the slowest test bed, more than a day is still too long to wait for the results of interpolating a single fMRI time series' data.

4.5 OpenCL

OpenCL (Open Computing Language) is an open standard designed by the Khronos group for the cross-platform parallel programming of processors found in computers, servers and embedded devices. OpenCL is based on the C99 programming language and facilitates the acceleration of a wide range of algorithms and programming patterns. The kernel for the parallelSinc interpolation algorithm was implemented in C++. Given the cross-platform nature of OpenCL, its performance was assessed on the CPU and GPU (graphics processing unit) hardware platforms independently.

4.5.1 OpenCL for CPU

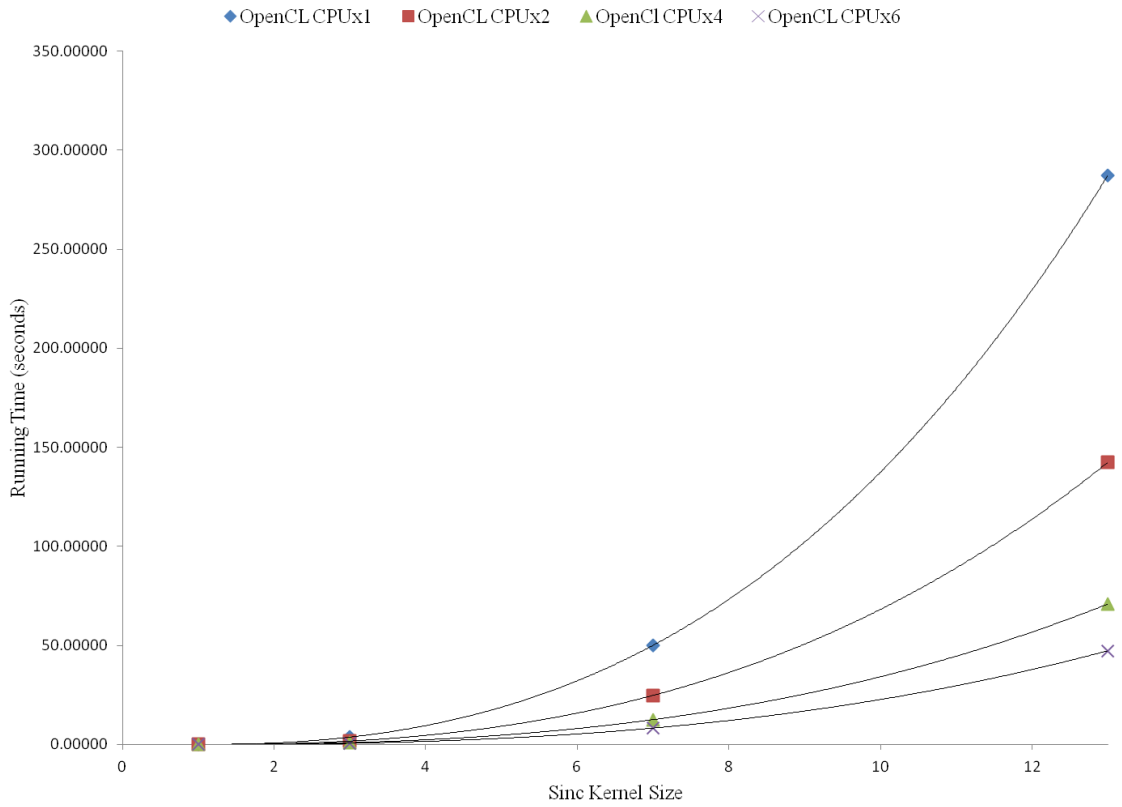

Figure 12: OpenCL CPU implementation running time comparison across enabled processor cores.

The results of comparing the performance of the OpenCL implementation of the parallelSinc algorithm for the CPU across four test hardware configurations are presented

in Figure 12. The raw data for this comparison are provided in Appendix E. Each running time presented is the average of 10 trials of executing the algorithm. Once again, for all test beds the running time increases as a cubic polynomial. This comparison serves to demonstrate the differences in execution time which can be achieved within the same processor (AMD Thuban 1090T clocked at 3221Mhz) based upon how many cores are available to run concurrent copies of the parallel kernel and the effect of overheads.

With a kernel size 13 x 13 x 13, running on two CPU cores the OpenCL parallelSinc implementation can interpolate the intensities of the 230400 voxels in an entire fMRI volume in approximately 142 seconds. Predictably, the best case performance is provided by running the CPU with all six cores enabled ($F(2,10) = 451706$, $p < 0.000001$); in this case, the same number of interpolations with the same sinc kernel size can be completed in 47 seconds. This is a speedup of approximately 3 times when the number of processor cores enabled is increased by a factor of 3.

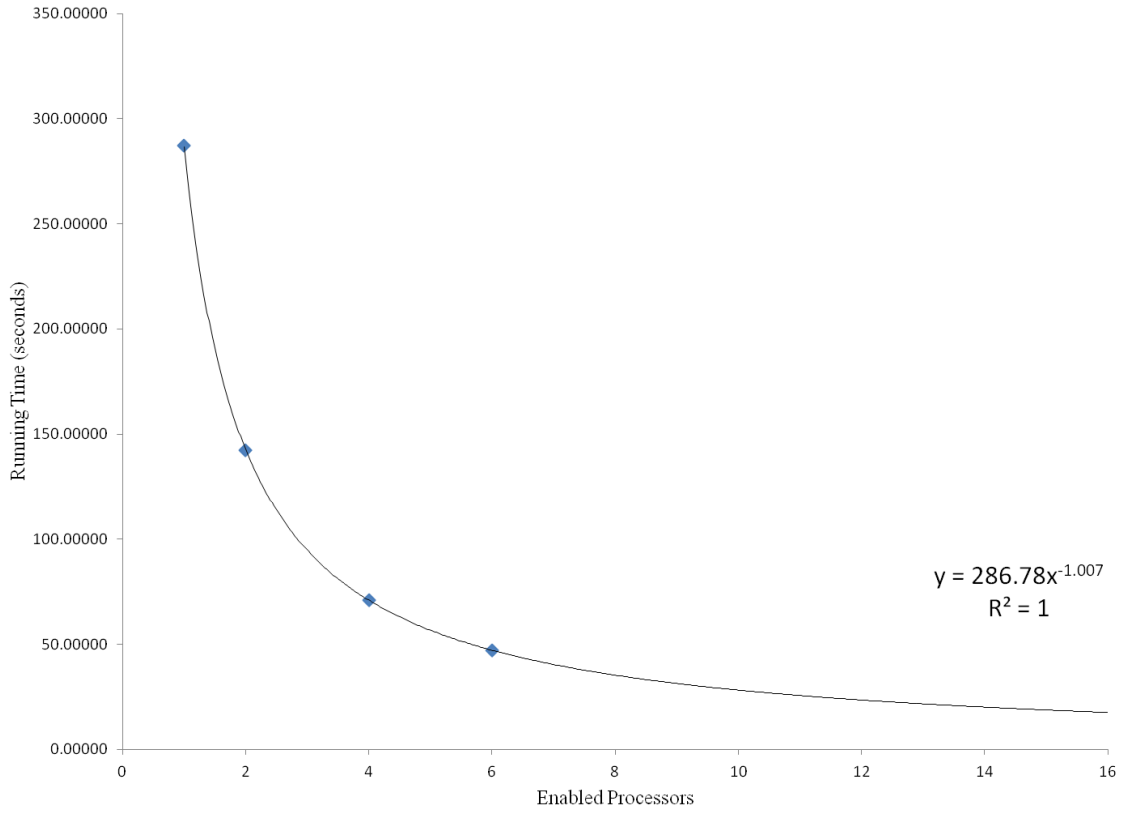

Figure 13: Running time as a function of enabled processor cores.

Figure 13 displays the power law relationship between running time and the number of enabled processor cores. Importantly, in concordance with Amdahl's law this relationship implies that the gains provided by an increasing number of CPU cores are asymptotic; that is to say, there is a limit to the improvement that can be provided by increasing the number of cores in a processor (given the same work load).

A demonstration of the overheads introduced by OpenCL can be seen by adding the running times of the C implementation of serialSinc on one core of the AMD Thuban 1090T clocked at 3221Mhz to Figure 12, shown in red with blue markers on Figure 14.

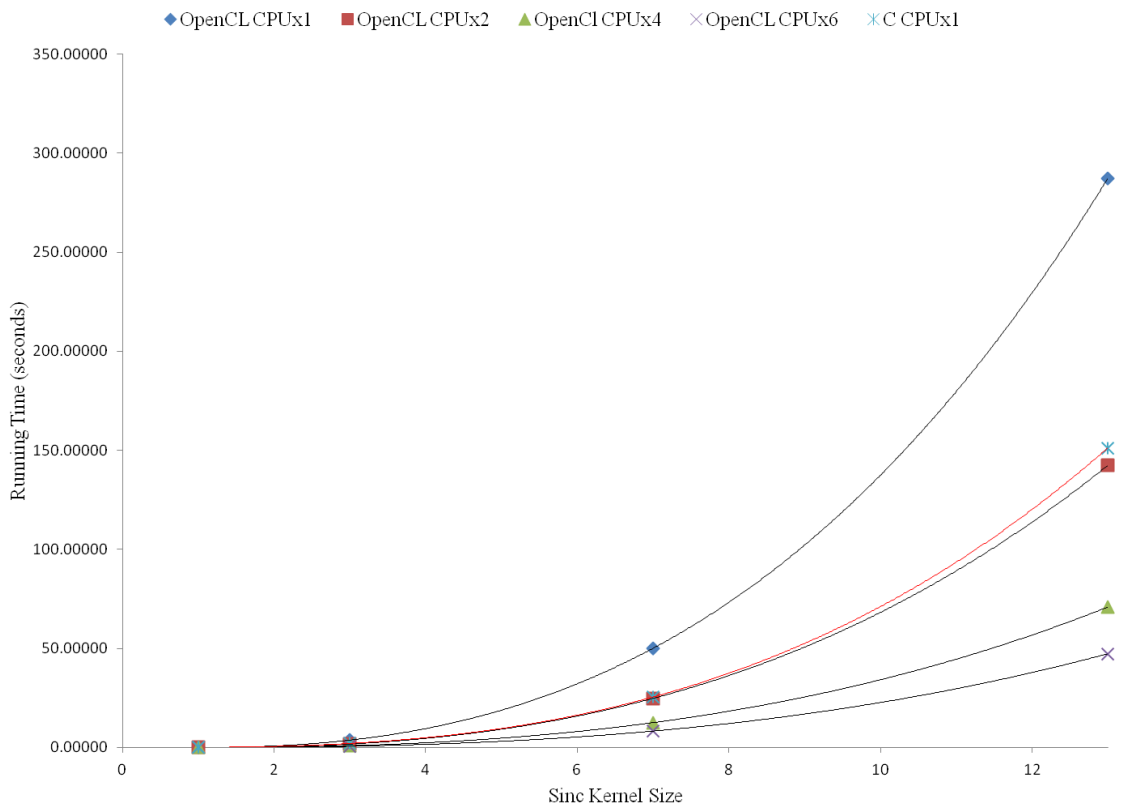

Figure 14: OpenCL CPU implementation running time comparison across enabled processor cores with single-core C implementation.

At 151 seconds, the performance of the serial C implementation is superior to the OpenCL implementation when there is only one processor core enabled ($F(2,10) = 52056$, $p < 0.000001$). This can be explained by the burdens introduced by the OpenCL

runtime (to handle thread switching, etc.) which are absent from the pure C implementation of the serialSinc 3D sinc interpolation algorithm. With two cores enabled, the OpenCL implementation for the CPU performs approximately as well as the C implementation, however still with a significant difference in running time ($F(2,10) = 545$, $p < 0.000001$). With four and six CPU cores enabled, the burden of parallelism is well compensated for by the concurrency provided by OpenCL and the OpenCL implementation outperforms the C implementation markedly.

The OpenCL implementation of 3D sinc interpolation for CPU also does not provide a tractable solution for performing 3D sinc interpolation with a kernel of size 13 x 13 x 13 for an entire fMRI time series. When using all six cores of the AMD Thuban 1090T processor, interpolating an entire volume takes 47 seconds and therefore it would take approximately 10.5 hours to interpolate an entire 800 volume time series; this is still not a desirable running time given that a single fMRI study could have more than 10 participants with several scanning sessions per participant.

4.5.2 OpenCL for GPU

The performance of the OpenCL implementation of the parallelSinc algorithm for the GPU is shown in Figure 15. The raw data for this implementation are provided in Appendix F. Each running time presented is the average of 10 trials of executing the algorithm; the running time increases as a cubic polynomial once again.

The OpenCL implementation of 3D sinc interpolation for the GPU finally provides a tractable solution for performing 3D sinc interpolation with a kernel of size 13 x 13 x 13 for an entire fMRI time series. Interpolating an entire volume with a sinc kernel of this size takes 1.51 seconds and therefore an entire 800 volume time series for a 20 minute scanning session (TR = 1.5 seconds) could be interpolated almost in real time at 21 minutes.

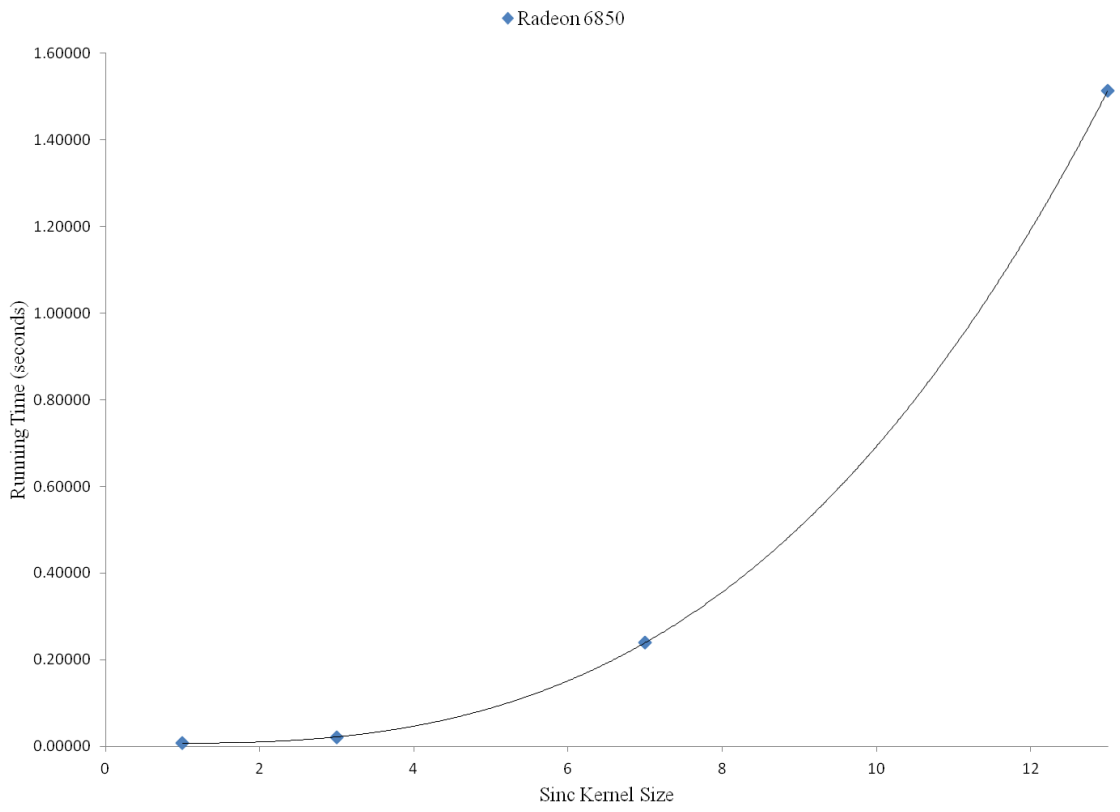

Figure 15: OpenCL GPU implementation running time performance.

4.6 Overall Results

Consider again the position of a neuroscience researcher who is trying to decide whether a given algorithm will be suitable to include in her neuroimaging data analysis pipeline, which is to run on her workstation desktop computer. In this section, the performance of the different languages explored previously is compared within the context of one computer system, test bed 1.

Presented in Figure 16 below are the performance results of benchmarking the 3D sinc interpolation algorithms described in Chapter 3 across programming language and hardware platform, within test bed 1. The results of benchmarking the Python implementation have been excluded from the figure for the sake of scale however they are listed with the other raw data for this figure in Appendix B.

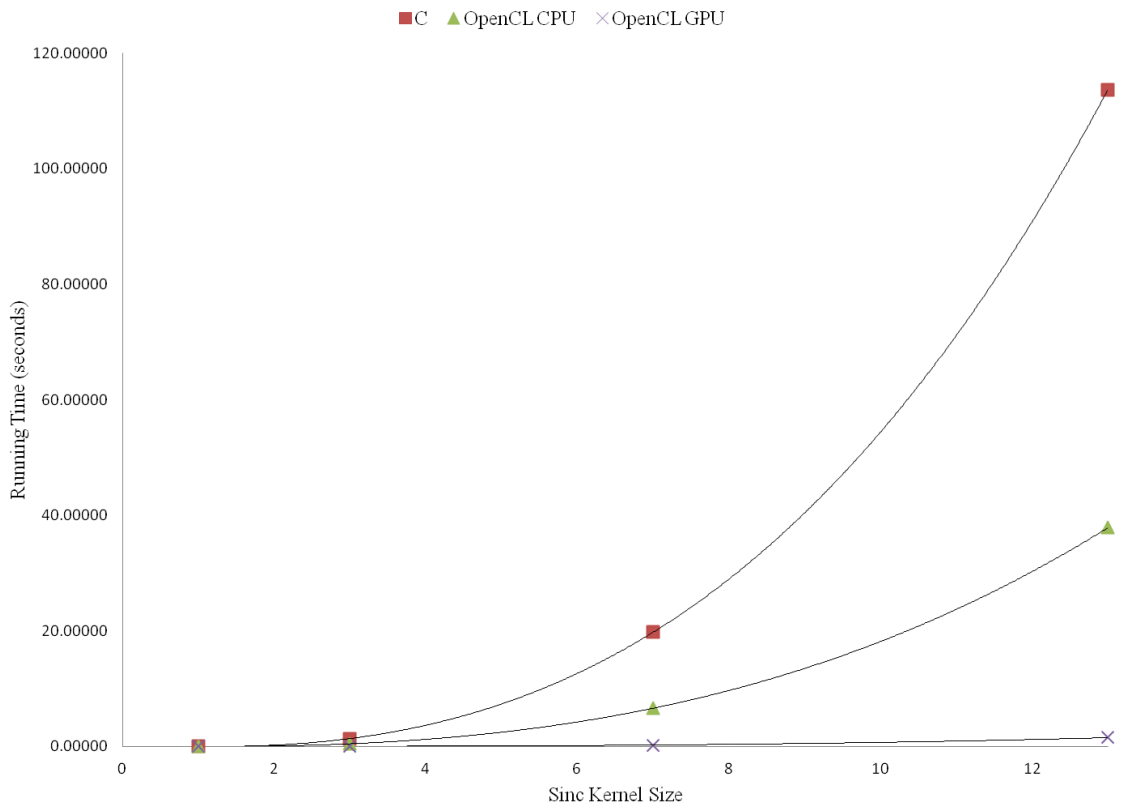

Figure 16: 3D sinc interpolation running time across programming language and hardware platform within test bed 1.

As evidenced by its exclusion from Figure 16 due to its excessively long running times, the Python implementation of the 3D sinc interpolation was by far the slowest of those tested with an estimated running time of 21200 seconds or 5.9 hours for an entire 80 x 80 x 36 voxel fMRI volume with a 13 x 13 x 13 sinc kernel. Python is an interpreted scripting language and does not benefit from the optimizations introduced by compilation, as do OpenCL and C. Specifically, the performance of an algorithm written in Python suffers severely when looping structures are used; the serialSinc algorithm uses 6 layers of nested FOR loops and as such the Python implementation performs accordingly.

The C implementation of the serialSinc algorithm was the next best in its running time performance, executing in approximately 114 seconds at an estimated 520 times faster rate than the Python implementation for the same sinc kernel with the same number of

interpolations. Although this version of the 3D sinc interpolation algorithm runs serially and only in a single thread, it is able to take advantage of the optimizations imbued by the C compiler when it is converted into a Python extension module by the distutils library. In general, the C programming language provides a high degree of performance for serial programs.

The OpenCL implementation of 3D sinc interpolation for CPU provides the next lowest running time at 37.8 seconds, providing approximately 3 times better performance than the C implementation and an estimated 1500 times better performance than the Python implementation for the same sinc kernel with the same number of interpolations ($F(2,10) = 185689$, $p < 0.000001$). In this implementation, the parallelSinc kernel is built and run on the CPU by the Python PyOpenCL package, taking advantage of the AMD Thuban 1090T processor's 6 cores to run 6 copies of the parallel kernel concurrently. Importantly, the PyOpenCL version of the algorithm does not run 6 times faster than the analogous C implementation, because of the overheads introduced by running the algorithm in parallel.

The OpenCL implementation of 3D sinc interpolation is improved substantially when run on the GPU, executing in slightly over 1.51 seconds for a sinc kernel of size 13 x 13 x 13 and vastly outperforming every other implementation: 39000 times faster than Python, 74 times faster than C, and 25 times faster than OpenCL for the CPU at 37.8 seconds ($F(2,10) = 1610875$, $p < 0.000001$). Graphics processing units contain hundreds of small processors called *stream processors*, which are designed to perform the simple mathematical operations used in rendering 3D graphics. The Radeon 6850 GPU in test bed 1 has 960 stream processors for this purpose; the OpenCL for GPU implementation is therefore able to take advantage of this hardware platform to run many hundreds of copies of the parallelSinc parallel kernel simultaneously. As such, the OpenCL for GPU implementation is the best option for performing 3D sinc interpolation available on this test bed.

Chapter 5

5 The Field Programmable Gate Array

A field programmable gate array (FPGA) is a customizable integrated circuit and can be used to compute the result of a complex logical function by physically implementing it within large blocks of reconfigurable logic gates. In this chapter I present the results of a novel implementation of the 3D sinc interpolation algorithm using an FPGA, beginning with a discussion of the hardware architecture of the test bed FPGA. Specifically, I compare the performance of two distinct parallel kernels for a sinc kernel size 13 x 13 x 13 and then assess the performance of an FPGA implementation of 3D sinc interpolation to the implementations described in Chapter 4.

5.1 Hardware Platform

Field programmable gate arrays are a specialized type of computer hardware which are reprogrammable and user customizable. FPGAs contain a combination of logic components and blocks of memory; these components can be used to implement digital computations in hardware within the FPGA. The hardware logic components are referred to as *logic blocks* and can be used to implement a wide range of combinational functions or serve as volatile memory elements.

The particular FPGA used to implement the 3D sinc interpolation algorithm described in Chapter 3 was the Altera Stratix V GS D5, housed in a Nallatech 385N PCIe computing card. This device has 8GB of onboard memory and communicates with the host computer over an 8-lane PCIe Gen 3 bus. The Altera Stratix V GS FPGA is optimized for high-performance, variable-precision digital signal processing (DSP) applications. This particular FPGA has 262400 adaptive logic modules, 3926 variable-precision DSP blocks, 2567 M20K memory blocks and 2 PCIe blocks, supporting a 14.1 Gbps host-device data rate; a schematic of the Altera Stratix V FPGA architecture and features is shown in Figure 17 below. The rack-mount server test bed described in Chapter 4 was used to host this FPGA device for performance testing.

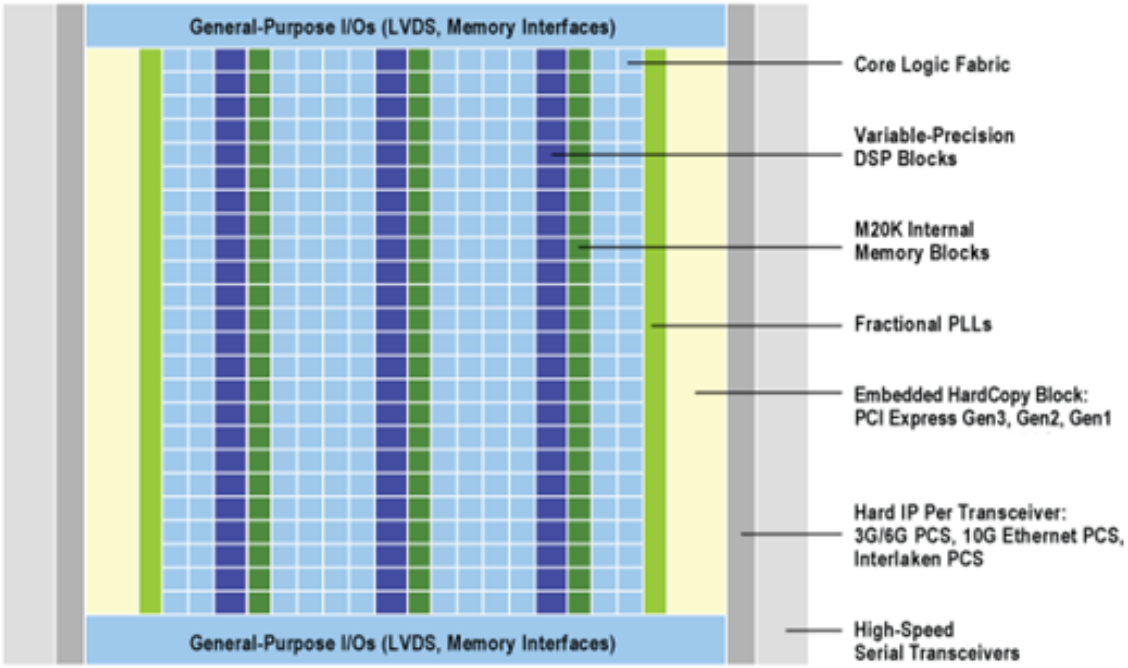

Figure 17: Altera Stratix V FPGA architecture and features (Altera Corporation).

5.2 Implementation

The same Python code to handle file I/O and performance testing as described in Chapter 4 was used for the FPGA implementation of 3D sinc interpolation. OpenCL host code to handle data transfer and execution of the parallel kernel on the FPGA was compiled using distutils as an extension to Python. The OpenCL parallel kernel code was built into a Raw Binary File (RBF) for the FPGA with the IBM Altera OpenCL compiler. The Altera Configuration via Protocol (CvP) system was then used to transfer the RBF and configure the logic components of the Altera Stratix V FPGA over the PCIe interface.

Two versions of the OpenCL parallelSinc kernel described in Chapter 3 were compiled for the FPGA. In the first version, the size of the sinc kernel used to perform the interpolation was dynamic and provided to the parallel kernel as a parameter. In the second version the size of the sinc kernel was static and fixed at 13 x 13 x 13, in an attempt to improve performance. The comparative utilization of the Altera Stratix V GS FPGA hardware components for these two versions is presented in Table 5 below.

Table 5: Dynamic versus static sinc kernel Altera Stratix V GS hardware utilization.

Resource	Dynamic Kernel	Static Kernel
Logic Utilization	31%	26%
Adaptive Lookup Table	17%	16%
Dedicated Logic Registers	14%	11%
Memory Blocks	29%	23%
DSP Blocks	54%	40%
Best Case FLOP Throughput	2687.84 MFLOPS	9250.00 MFLOPS

5.3 Performance Benchmarking

The results of a performance comparison of the dynamic and static 3D sinc interpolation kernels across input sizes with a sinc kernel of size 13 x 13 x 13 are shown in Figure 18. The raw data for this comparison are provided in Appendix G.

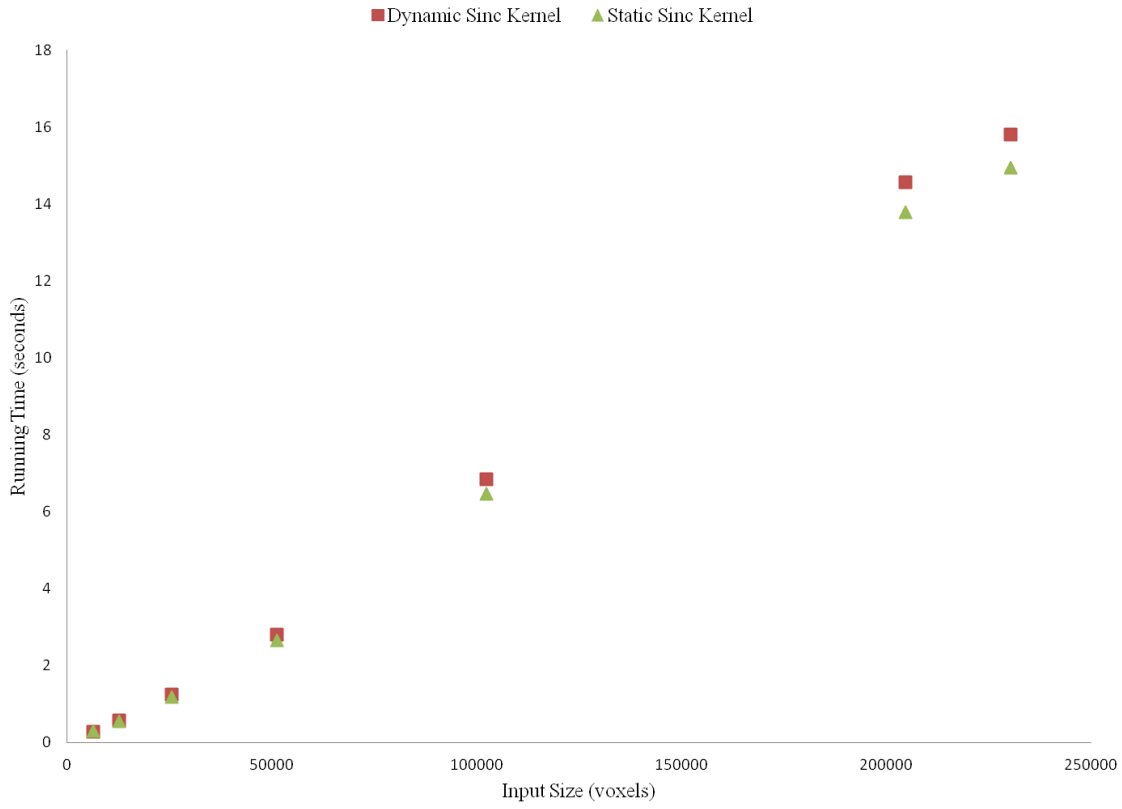

Figure 18: FPGA dynamic and static sinc kernel running time performance comparison for kernel size 13 x 13 x 13.

When interpolating the voxel intensities of an entire fMRI volume (230400 total interpolations), with a running time of 15.8 seconds the dynamic sinc kernel performed significantly worse than the static sinc kernel which had mean running time of 14.9 seconds ($F(2,10) = 78492.81$, $p < 0.000001$). This difference in performance can be explained by the optimizations offered by the IBM Altera OpenCL compiler when the number of loop iterations is predetermined as in the static sinc kernel parallel kernel.

As shown in Figure 19 below, to interpolate an entire 230400 voxel fMRI volume with a 3D sinc kernel of size 13 x 13 x 13, at 15.8 seconds the dynamic FPGA implementation performed significantly better than the OpenCL implementation for CPU which took 37.8 seconds ($F(2,10) = 590851$, $p < 0.000001$). The FPGA implementation however performed worse than the OpenCL GPU implementation of the parallelSinc kernel at 1.51 seconds ($F(2,10) = 62851220$, $p < 0.000001$).

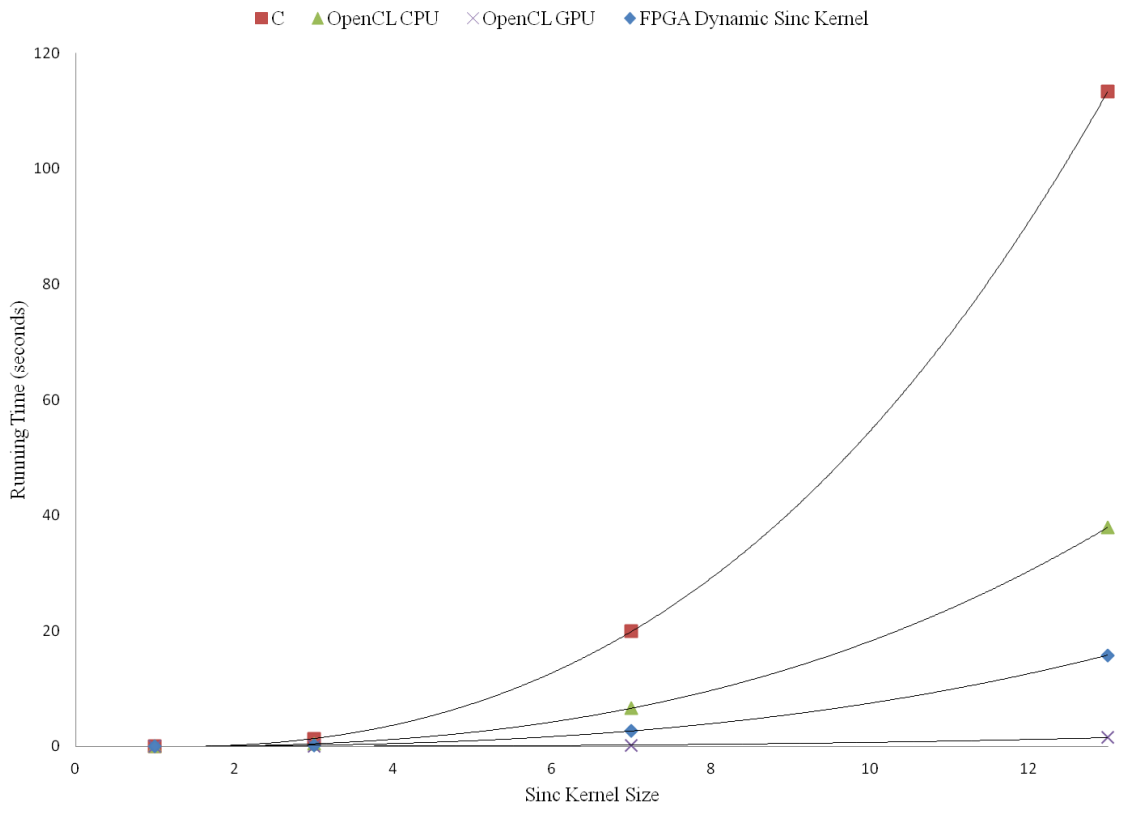

Figure 19: 3D sinc interpolation running time across programming language and hardware platform within test bed 1, including the FPGA dynamic sinc kernel.

The FPGA is also a reasonable platform for performing 3D sinc interpolation with a kernel of size 13 x 13 x 13 for an entire fMRI series. Interpolating an entire volume with a sinc kernel of this size takes 15.8 seconds and so the example 800 volume fMRI time (20 minutes, TR = 1.5 seconds) series from Chapter 4 could be interpolated in 3.5 hours.

5.4 Power Considerations

Although running the 3D sinc interpolation on the FPGA cannot provide performance surpassing that of the GPU, an important consideration is the amount of power used to complete the computation. Under load, a powerful GPU might draw between 300-500W of power. The most power-hungry FPGA however, will draw at most 10W of power under load. Consider a large fMRI study, in which each of 20 participants has 1500 volumes of scanner data. The energy required by the device to interpolate the raw data for this study with the 500W Radeon 6850 in test bed 1 is

$$\frac{500\ joules}{second} \times \frac{1.51 seconds}{1\ volume} \times \frac{1500\ volumes}{participant} \times 20\ participants = 6.292 kWh$$

whereas the energy required to interpolate the same dataset with the 4W Altera Stratix V FPGA housed in a Nallatech 385N card is

$$\frac{4\ joules}{second} \times \frac{15.79 seconds}{1\ volume} \times \frac{1500\ volumes}{participant} \times 20\ participants = 0.5263 kWh$$

This is to say, the same fMRI preprocessing task could be completed with an FPGA using less than 10% of the power of a GPU implementation. Given that an average household consumes 30kWh of energy per day (U.S. Energy Information Administration), the energy savings afforded by the use of a FPGA are meaningful.

Chapter 6

6 Case Study: Robust Motion Correction

Current state of the art motion correction algorithms can adequately compensate for the head movements in a typical subject, however fail to correct for the greater head movements in special populations. Conor Wild and Rhodri Cusack developed a new motion correction algorithm which can compensate for these greater head movements (personal communication, May 15, 2013). In this chapter, I show the performance benefits which can be realized in Wild and Cusack's motion correction algorithm by improving the implementation of the interpolation step.

6.1 Algorithm

Traditional motion correction algorithms such as those outlined in Chapter 2 are designed to handle the small (e.g. 1mm) movements which occur in normal subjects. Special patient populations such as the elderly, mentally ill, or infants can however produce movements of up to several millimeters during a scanning session (Friston et al., 1996). Traditional motion correction algorithms can fail when applied to scanning data from subjects who display these greater magnitudes of motion; these failures would be due to an inability of the optimization algorithm to find a global minimum for the error function given the high degree of error present at the outset of optimization.

As such, there is a need for more robust motion correction algorithms which can accommodate these populations if they are to be studied with traditional fMRI paradigms. In response to this need, Wild and Cusack (personal communication, May 15, 2013) developed a general motion correction algorithm which can provide for robust motion correction (i.e. correcting for subject movements greater than 1mm) as well as traditional motion correction (i.e. correcting for subject movements less than 1mm). This algorithm, henceforth referred to as RMC, is similar in structure to the FSL motion correction algorithm described in Chapter 2. A flowchart of the RMC algorithm is shown in Figure 20 below.

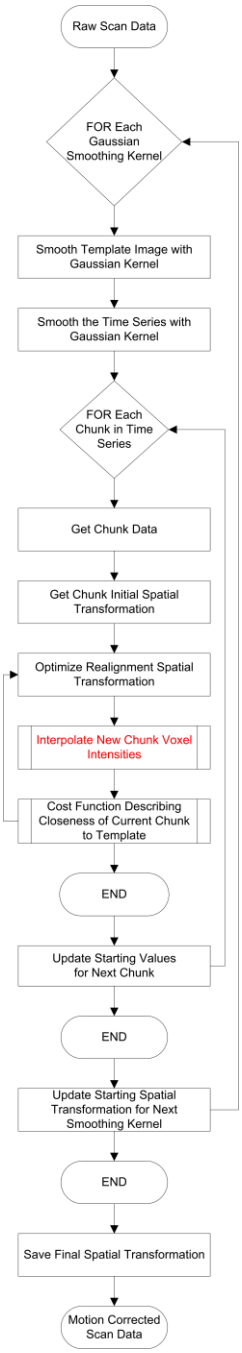

Figure 20: Wild and Cusack's robust motion correction algorithm flowchart.

The RMC algorithm uses a variation of the Global-Local optimization proposed by Jenkinson, Bannister, Brady and Smith (2001). For each level of smoothing (performed with a Gaussian smoothing kernel), for each *chunk* in the time series (a chunk is a collection of slices, up to an entire volume of slices), the RMC algorithm uses an

optimization algorithm to find a solution spatial transformation which reduces the difference between that chunk and the corresponding chunk in a template image. In this algorithm, the rigid body assumption for the entire brain is violated, however it is still preserved for each chunk of slices.

The optimization algorithm begins its search for the solution spatial transformation of a given chunk with a linear combination of the solution transformation from the previous chunk at the current smoothing level and the solution transformation from the current chunk at the previous smoothing level. The total number of optimizations performed is therefore given by the number of smoothing levels multiplied by the number of chunks in the time series. Within each of these optimizations, for each new potential spatial transformation the voxel intensities for the entire chunk must be interpolated from the raw scan data in order to evaluate the cost (error) function. Once the spatial transformation for every chunk in the time series has been computed for the final smoothing level, these parameters are saved and the corrected time series data are interpolated from the raw scan data.

6.2 Performance Benchmarking

As suggested in the previous section, the RMC algorithm needs to perform many 3D interpolations to determine the realignment parameters for an entire fMRI time series (interpolation is shown in red on Figure 20). The number of interpolations necessary to determine the final solution spatial transformation for an entire time series is given by

$$[smoothing\ levels] \times [chunks] \times [optimization\ iterations]\ \times\ [voxels\ per\ chunk]$$

and therefore the performance of the interpolation method used in RMC is of central importance to its performance as a whole. For this reason, the RMC algorithm is an ideal practical example of the performance gain which can be realized by improving the implementation of the interpolation step.

To demonstrate this performance gain, the running times to correct the motion in one 80 x 80 x 36 voxel fMRI volume for two versions of the RMC algorithm were compared: one version of the RMC algorithm used the C implementation of the serialSinc algorithm

running on test bed one's AMD Thuban 1090T CPU; the other version used the OpenCL implementation of the parallelSinc algorithm running on test bed one's Radeon 6850 GPU. In both of these versions of the RMC algorithm three smoothing levels were used (with Gaussian smoothing kernel sizes of 4, 2 and 1) and the chunk size was 6 slices, meaning there were 6 chunks in total. To perform the optimization step, Powell optimization was used with a sum of squared error cost function. The results of this comparison are presented in Figure 21 below.

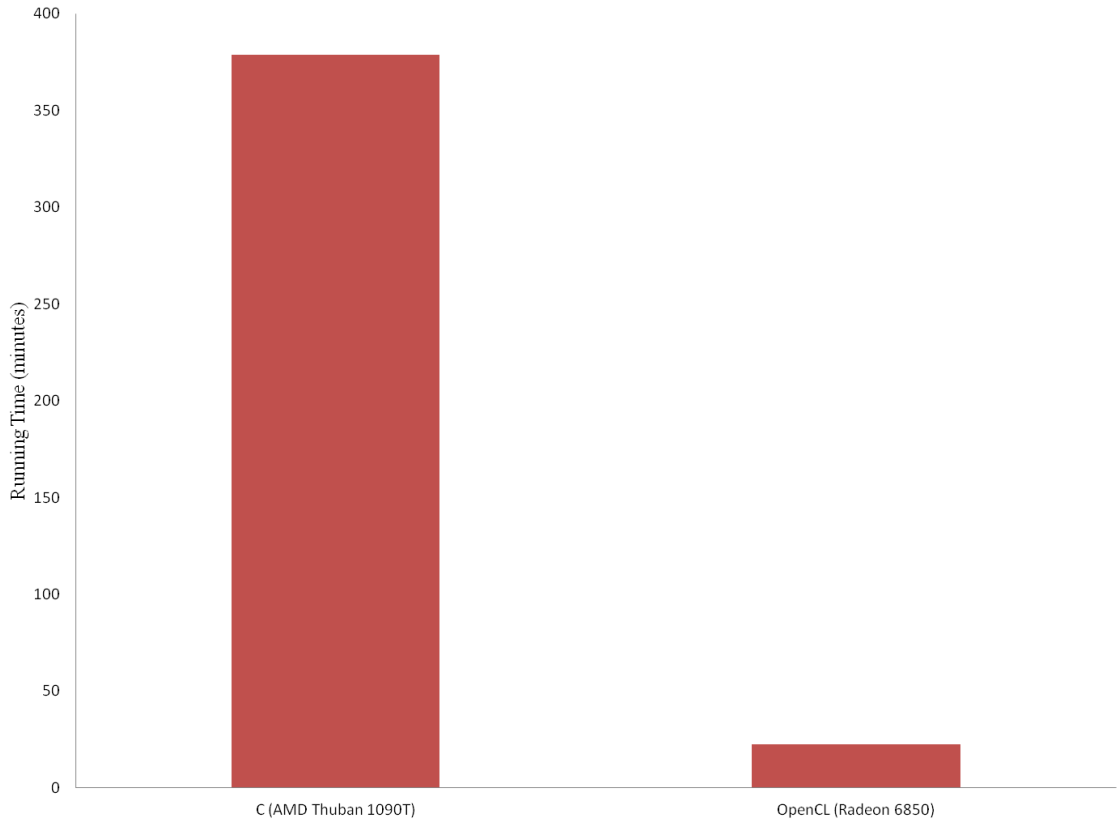

Figure 21: RMC robust motion correction algorithm performance comparison within test bed 1 for C and OpenCL for GPU.

The version of the RMC algorithm using a C implementation of 3D sinc interpolation had a running time of 379 minutes, whereas the version of RMC using an OpenCL implementation had a running time of 22.6 minutes, one sixteenth that of the C implementation.

Chapter 7

7 Conclusions

The focus of the present thesis has been on comparing the performance of different implementations of 3D sinc interpolation across multiple hardware platforms and programming languages. The results of these comparisons indicate a progressive improvement in performance from Python, C, OpenCL for CPU and OpenCL for GPU. Although the asymptotic time complexity of the algorithm might be equal among several of these implementations, the relative performance of different implementations is important because for neuroimaging data processing algorithms there is a hard constraint on the input size imposed by the physical limitations of the fMRI scanner and the physiological limitations of the subject.

Moreover, although the results of this performance comparison might seem easily predictable to an individual versed in parallel programming techniques specifically or software engineering generally, they do serve to illustrate an important point: before an accurate algorithm (in this case sinc interpolation) is dismissed as unfeasible for the solution of a particular problem (i.e. 3D fMRI data interpolation) because of long running times on one test bed, alternatives for the hardware platform and software language used in its implementation should be considered.

In addition to these comparisons, in this thesis a novel implementation of 3D sinc interpolation on a field programmable gate array was presented. This implementation performed better than an OpenCL implementation running on a 6 core CPU, however it performed worse than an OpenCL implementation running on a GPU. Although the FPGA was unable to offer better performance than the existing GPU OpenCL implementation, it was shown to consume substantially less power in completing the same computations.

Finally, the performance of an entire algorithm for robust fMRI motion correction employing an implementation of 3D sinc interpolation in C and an implementation using OpenCL for the GPU were compared. These results again showed the superiority of the

GPU implementation of the algorithm, however this time in the context of a practical fMRI data preprocessing algorithm.

7.1 Threats to Validity

One threat to the validity of the results presented herein is the difference in the Python clock utility used to measure running time across the multiple operating systems used on the four experimental test beds. On Windows, the Python clock() function returns as a floating point number the wall-clock seconds elapsed since the first call to the function based on the Win32 function QueryPerformanceCounter() with microsecond accuracy. On Unix, the python clock() function returns the current processor time as a floating point number expressed in seconds, with the precision depending on the eponymous C function. Despite these differences the Python documentation indicates that regardless of the operating system, clock() is the function to use for benchmarking Python or timing algorithms.

Another general criticism of the results presented could stem from the differences in I/O handling, floating point number handling, or compiler optimization between the multiple test beds described in Chapter 4. A critic could argue that because of these differences, comparisons of the performance of the 3D sinc interpolation between these systems are invalid, because of the lack of consistency between the systems. The comparisons presented were however intended not to rigorously compare the performance of the different CPU hardware architectures in each test bed, but instead to highlight the differences between each test bed as a whole.

Another criticism could stem from the repeated paired statistical comparisons presented to the exclusion of a stricter multiple comparisons test such as Tukey's Honestly Significant Difference test. Multiple comparison tests such as Tukey's, however, typically involve only setting a stricter threshold for the significance of the *p* value reported to produce a new 'effective value' which compensates for the higher probability of a type I error present when making multiple paired comparisons. The statistical significance of the difference between each pair of sample means presented was however so powerful

that even with an extremely strict threshold for significance, each comparison would certainly achieve significance and allow for the rejection of the null hypothesis.

7.2 Future Directions

There are several future directions for this research. Firstly, comparisons between different GPU and FPGA hardware platforms could be undertaken. At present, the performance of the 3D sinc interpolation kernel was only evaluated on one GPU (the Radeon 6850) and one FPGA device (the Altera Stratix V GS). Benchmarking the algorithm's performance on a GPU from a different manufacturer such as nVIDIA, or a FPGA from a different manufacturer such as Xilinx could provide interesting results. For example, a state of the art GPU like the nVIDIA Titan might be able to perform 3D sinc interpolation with a sinc kernel size 13 x 13 x 13 in real time for a scanner with a volume resolution of 80 x 80 x 36 voxels.

As explained in Chapter 3, the 3D sinc interpolation algorithm was parallelized only at the level of the voxels in a given chunk. Parallelizing the sinc interpolation algorithm with a smaller granularity, such as to the level of computing the sinc contribution of each voxel in a sinc kernel concurrently, could enhance performance of the parallel version of the algorithm further.

Finally, although the 3D sinc interpolation algorithm is an interesting and valuable example for comparing the relative performance of different combinations of programming language and hardware platform, there are several other 3D interpolation methods commonly used in fMRI motion correction which might be worthy of consideration. It could be interesting and fruitful to compare the performance of different combinations of programming language and hardware platform for Fourier interpolation as well as trilinear interpolation in a manner similar to that presented herein.

References

Amdahl, G. M. (1967, April). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (pp. 483-485). ACM.

Bandettini, P. A., Wong, E. C., Hinks, R. S., Tikofsky, R. S., Hyde J. S. (1992). Time course EPI of human brain function during task activation. *Magnetic Resonance in Medicine*, *25*, 390-397.

Che, S., Li, J., Sheaffer, J. W., Skadron, K., & Lach, J. (2008, June). Accelerating compute-intensive applications with GPUs and FPGAs. In *Application Specific Processors, 2008. SASP 2008. Symposium on* (pp. 101-107). IEEE.

Cox, R. W. (1996). AFNI: Software for analysis and visualization of functional magnetic resonance neuroimages. *Computers and Biomedical Research, 29*, 162-173.

Cox, R. W., & Jesmanowicz, A. (1999). Real-time 3D image registration for functional MRI. *Magnetic Resonance in Medicine, 42*, 1014-1018.

Eddy, W. F., Fitzgerald, M., & Noll, D. C. (1996). Improved imaged registration by using Fourier interpolation. *Magnetic Resonance in Medicine*, *36*, 923-931.

Field, A. S., Yen, Y. F., Burdette, J. H., & Elster, A. D. (2000). False cerebral activation on BOLD functional MR images: Study of low-amplitude motion weakly correlated to stimulus. *American Journal of Neuroradiology, 21*, 1388-1396.

Fornberg, B., & Zuev, J. (2007). The Runge phenomenon and spatially variable shape parameters in RBF interpolation. *Computers & Mathematics with Applications, 54*(3), 379-398.

Freire, L., & Mangin, J. F. (2000). Motion correction algorithms may create spurious brain activations in the absence of subject motion. *NeuroImage*, *14*, 709-722. doi: 10.1006/nimg.2001.0869

Friston, K. J., Ashburner, J., Frith, C. D., Poline, J. B., Heather, J. D. (1995). Spatial normalization and registration of images, *Human Brain Mapping*, *3*, 165-189.

Friston, K. J., Williams, S., Howard, R., Frackowiak, R. S. J., Turner, R. (1996). Movement-related effects in fMRI time series. *Magnetic Resonance in Medicine, 35*, 346-355.

Friston, K. J., Rotshtein, P., Geng, J. J., Sterzer, P., Henson, R. N. (2006). A critique of functional localisers. *NeuroImage*, *30*, 1077-1087.

Glover, G. H., Li, T. Q., & Ress, D. (2000). Image-based method for retrospective correction of physiological motion effects in fMRI: RETROICOR. *Magnetic Resonance in Medicine*, *44*(1), 162-167.

Hajnal, J. V., Saeed, N., Soar, E. J., Oatridge, A., Young, I. R., & Bydder, G. M. (1995). A registration and interpolation procedure for subvoxel matching of serially acquired MR images. *Journal of Computer Assisted Tomography*, *19*(2), 289-296.

Hu, X., Le, T. H., Parrish, T., & Erhard, P. (1995). Retrospective estimation and correction of physiological fluctuation in functional MRI. *Magnetic Resonance in Medicine*, *34*, 201-212.

Huettel, S. A., Song, A. W., & McCarthy, G. (2004). *Functional magnetic resonance imaging*. Sinauer Associates.

Jenkinson, M., Bannister, P., Brady, M., & Smith, S. (2001). Improved optimization for the robust and accurate linear registration and motion correction of brain images. *NeuroImage*, *17*, 825-841. doi: 10.1006/nimg.2002.1132

Jiang, A., Kennedy, D. N., Baker, J. R., Weisskoff, R. M., Tootell, R. B. H., Woods, R.P., Benson, R. R., Kwong, K. K., Brady, T. J., Rosen, B. R., Belliveau, J. W. (1995). Motion detection and correction in functional MR imaging. *Human Brain Mapping*, *3*, 224–235. doi: 10.1002/hbm.460030306

Kostelec, P. J., & Periaswamy, S. (2003). Image registration for MRI. *Modern Signal Processing*, *46*, 161-184.

Lemieux, L., Salek-Haddadi, A., Lund, T. E., Laufs, H., Carmichael, D. (2007). Modelling large motion events in fMRI studies of patients with epilepsy. *Magnetic Resonance Imaging*, *25*, 894-901.

Lund, T. E., Nørgaard, M. D., Rostrup, E., Rowe, J. B., Paulson, O. B. (2005). Motion or activity: Their role in intra- and inter-subject variation in fMRI. *NeuroImage*, *26*, 960-964.

Noll, D. C., & Schneider, W. (1994). Theory, simulation and compensation of physiological motion artifacts in functional MRI. *Proceedings of IEEE International Conference on Image Processing*, 40.

Oakes, T. R., Johnstone, T., Ores Walsh, K. S., Greischar, L. L., Alexander, A. L., Fox, A. S., & Davidson, R. J. (2005). Comparison of fMRI motion correction software tools. *NeuroImage*, *28*(3), 529-543.

Ogawa, S., Tank, D. W., Menon, R., Ellermann, J., Kim, S. G., Merkle, H., Ugurbil, K. (1992). Intrinsic signal changes accompanying sensory stimulation: Functional brain mapping with magnetic resonance imaging. *Proceedings of the National Academy of Sciences*, *89*, 5951-5955.

Ooi, M. B., Krueger, S., Thomas, W. J., Swaminathan, S. V., & Brown, T. R. (2009). Prospective real-time correction for arbitrary head motion using active markers. *Magnetic Resonance in Medicine*, *62*(4), 943-954.

Oppenheim, A. V. (1975). RW Schafer 'Digital Signal Processing'. *Prentice-Hall, Englewood Cliffs, New Jersey*, *6*, 125-136.

Powell, M. J. D. (1964). An efficient method for finding the minimum of a function of several variables without calculating derivatives. *Computer Journal*, *7*, 155-162. doi: 10.1093/comjnl/7.2.155

Speck, O., Hennig, J., & Zaitsev, M. (2006). Prospective real-time slice-by-slice motion correction for fMRI in freely moving subjects. *Magnetic Resonance Materials in Physics, Biology and Medicine*, *19*(2), 55-61.

Strother, S. C. (2006). Evaluating fMRI preprocessing pipelines: Review of preprocessing steps for BOLD fMRI. *IEEE Engineering in Medicine and Biology Magazine*, 27- 41. doi: 0739-5175

Thacker, N. A., Jackson, A., Moriarty, D., & Vokurka, E. (1999). Improved quality of re-sliced mr images using re-normalized sinc interpolation. *Journal of Magnetic Resonance Imaging*, *10*, 582-588.

Thesen, S., Heid, O., Mueller, E., & Schad, L. R. (2000). Prospective acquisition correction for head motion with image-based tracking for real-time fMRI. *Magnetic Resonance in Medicine*, *44*, 457-465.

Tong, R., & Cox, R. W. (1999). Rotation of NMR images using the 2D chirp-z transform. *Magnetic Resonance in Medicine*, *41*(2), 253-256.

Tremblay, M., Tam, F., & Graham, S. J. (2005). Retrospective coregistration of functional magnetic resonance imaging data using external monitoring. *Magnetic Resonance in Medicine*, *53*(1), 141-149.

Twieg, D. (1983). The k-trajectory formulation of the NMR imaging process with applications in analysis and synthesis of imaging methods. *Medical Physics*, *10*, 610-21. doi: 10.1118/1.595331

Webster, R., & Oliver, M. A. (2001). Geostatistics for environmental scientists (statistics in practice).

Weiskopf, N., Sitaram, R., Josephs, O., Veit, R., Scharnowski, F., Goebel, R., ... & Mathiak, K. (2007). Real-time functional magnetic resonance imaging: methods and applications. *Magnetic Resonance Imaging*, *25*(6), 989-1003.

Woods, R. P., Cherry, S. R., & Mazziotta, J. C. (1992). Rapid automated algorithm for aligning and reslicing PET images. *Journal of Computer Assisted Tomography*, *16*(4), 620-633.

Woods, R. P., Grafton, S. T., Holmes, C. J., Cherry, S. R., & Mazziotta, J. C. (1998). Automated image registration: I. General methods and intrasubject, intramodality validation. *Journal of Computer Assisted Tomography*, *22*(1), 139-152.

Zaitsev, M., Dold, C., Sakas, G., Hennig, J., & Speck, O. (2006). Magnetic resonance imaging of freely moving objects: prospective real-time motion correction using an external optical motion tracking system. *Neuroimage*, *31*(3), 1038-1050.

Appendices

Appendix A: List of software packages and version numbers.

Test Bed 1 - Workstation Desktop	
C Compiler	Microsoft (R) C/C++ 15.00.21022.08 for x64
Python	2.7
PyOpenCL	2012.1
Numpy	1.7.0
Scipy	0.12.0b1
Nibabel	1.3.0
matplotlib	1.2.0
AMD VISION Engine	13.4
Test Bed 2 - Laboratory Desktop	
C Compiler	GCC 4.7.3
Python	2.7.4
Numpy	1.7.1
Scipy	0.11.0
Test Bed 3 - Rack-Mount Server	
C Compiler	GCC 4.4.6
Python	2.7.5
Numpy	1.7.1
Scipy	1.12.0
Test Bed 4 - Legacy Laptop	
C Compiler	(same as test bed 1)
Python	2.7.3
Numpy	1.7.0
Scipy	0.11.0
Nibabel	1.3.0
matplotlib	1.3.0

Appendix B: 3D sinc interpolation mean running time across programming language and hardware platform within test bed 1 data table.

Kernel Size	Slices Per Chunk	Voxels	Python	C	OpenCL CPU	OpenCL GPU
1	1	6400	0.09905	0.00080	0.00197	0.00250
3	1	6400	5.44586	0.02354	0.01024	0.00317
7	1	6400	79.9558	0.32574	0.11669	0.00721
13	1	6400	N/A	1.97805	0.70556	0.03045
1	2	12800	0.18917	0.00113	0.00208	0.00226
3	2	12800	12.6758	0.05455	0.02205	0.00365
7	2	12800	171.779	0.69705	0.24651	0.01154
13	2	12800	N/A	4.10875	1.47251	0.05827
1	4	25600	0.37508	0.00199	0.00199	0.00258
3	4	25600	31.0592	0.13195	0.05065	0.00442
7	4	25600	385.869	1.58344	0.55282	0.02453
13	4	25600	N/A	8.88442	3.15047	0.12450
1	8	51200	0.75490	0.00403	0.00309	0.00303
3	8	51200	68.1876	0.28854	0.10095	0.00681
7	8	51200	936.254	3.87883	1.32138	0.06046
13	8	51200	N/A	19.9882	6.97667	0.28417
1	16	102400	1.47991	0.00741	0.00512	0.00449
3	16	102400	140.685	0.59934	0.20391	0.01125
7	16	102400	2120.19	8.69777	2.92332	0.10882
13	16	102400	N/A	49.0161	16.4872	0.68811
1	32	204800	2.96186	0.01533	0.00892	0.00735
3	32	204800	290.354	1.22223	0.41207	0.02019
7	32	204800	4435.09	18.2435	6.08879	0.21176
13	32	204800	N/A	104.408	35.0277	1.34306
1	36	230400	3.33900	0.01705	0.00975	0.00764
3	36	230400	317.506	1.35719	0.45108	0.02197
7	36	230400	4862.61	19.7572	6.60705	0.23935
13	36	230400	N/A	113.635	37.8461	1.51343

Appendix C: Python implementation mean running time comparison across hardware architecture data table.

Kernel Size	Slices Per Chunk	Chunk Size	CPU: AMD Turion TL-56	CPU: AMD Thuban 1090T	CPU: Intel Xeon E5603	CPU: Intel Xeon E5504
1	1	6400	0.30799	0.09905	0.24420	0.14320
3	1	6400	15.6981	5.44586	15.4500	10.1600
7	1	6400	224.373	79.9558	225.480	149.180
1	2	12800	0.54832	0.18917	0.46700	0.28080
3	2	12800	35.0594	12.6758	36.5600	23.6300
7	2	12800	475.025	171.779	481.920	320.550
1	4	25600	1.07633	0.37508	0.94340	0.55240
3	4	25600	85.3394	31.0592	88.5900	56.4500
7	4	25600	1077.04	385.869	1093.40	724.920
1	8	51200	2.13736	0.75490	1.86760	1.09800
3	8	51200	186.212	68.1876	191.470	125.160
7	8	51200	2621.09	936.254	2638.18	1770.90
1	16	102400	4.27821	1.47991	3.72680	2.19160
3	16	102400	389.879	140.685	394.600	260.300
7	16	102400	5946.47	2120.19	6028.40	3995.17
1	32	204800	8.46581	2.96186	7.40300	4.37940
3	32	204800	789.531	290.354	815.000	521.650
7	32	204800	12306.4	4435.09	12644.3	8387.59
1	36	230400	9.29165	3.33900	8.19300	5.14100
3	36	230400	858.706	317.506	898.255	581.038
7	36	230400	13147.54	4862.61	13647.1	9069.40

Appendix D: C implementation mean running time comparison across hardware architecture data table.

Kernel Size	Slices Per Chunk	Voxels	CPU: AMD Turion TL-56	CPU: AMD Thuban 1090T	CPU: Intel Xeon E5603	CPU: Intel Xeon E5504
1	1	6400	0.00227	0.00080	0.00100	0.00060
3	1	6400	0.05974	0.02354	0.05180	0.10400
7	1	6400	0.81697	0.32574	0.82080	1.55460
13	1	6400	5.14122	1.97805	5.58380	9.83620
1	2	12800	0.00342	0.00113	0.00180	0.00120
3	2	12800	0.13578	0.05455	0.12060	0.24200
7	2	12800	1.74915	0.69705	1.75920	3.33060
13	2	12800	10.6442	4.10875	11.5958	20.4360
1	4	25600	0.00662	0.00199	0.00280	0.00200
3	4	25600	0.32599	0.13195	0.29220	0.58580
7	4	25600	3.95834	1.58344	3.98700	7.54860
13	4	25600	22.8674	8.88442	24.9098	43.8206
1	8	51200	0.01129	0.00403	0.00520	0.00460
3	8	51200	0.70593	0.28854	0.63580	1.27660
7	8	51200	9.66314	3.87883	9.73100	18.4126
13	8	51200	52.0704	19.9882	56.6878	99.6924
1	16	102400	0.02066	0.00741	0.00980	0.00680
3	16	102400	1.46545	0.59934	1.32300	2.65460
7	16	102400	21.7726	8.69777	21.9184	41.5054
13	16	102400	126.938	49.0161	138.281	243.587
1	32	204800	0.03912	0.01533	0.01940	0.01380
3	32	204800	2.98630	1.22223	2.69700	5.41220
7	32	204800	45.5416	18.2435	45.9606	87.1916
13	32	204800	270.240	104.408	294.574	517.887
1	36	230400	0.04332	0.01705	0.02200	0.01500
3	36	230400	3.29573	1.35719	2.98300	6.03700
7	36	230400	49.7613	19.7572	49.9390	93.5930
13	36	230400	292.338	113.635	310.316	561.970

Appendix E: OpenCL CPU implementation mean running time comparison across enabled processor cores data table.

Kernel Size	Slices Per Chunk	Voxels	OpenCL CPUx1	OpenCL CPUx2	OpenCl CPUx4	OpenCL CPUx6
1	1	6400	0.00271	0.00214	0.00255	0.00213
3	1	6400	0.06272	0.03051	0.01686	0.01252
7	1	6400	0.81735	0.41134	0.21402	0.14394
13	1	6400	4.97512	2.48294	1.27170	0.88373
1	2	12800	0.00312	0.00239	0.00275	0.00220
3	2	12800	0.13687	0.07001	0.03683	0.02771
7	2	12800	1.75404	0.87296	0.45412	0.30744
13	2	12800	10.3356	5.15059	2.63962	1.82183
1	4	25600	0.00455	0.00364	0.00323	0.00310
3	4	25600	0.32941	0.16878	0.08977	0.06260
7	4	25600	3.98209	2.01137	1.02545	0.68966
13	4	25600	22.4482	11.3126	5.82611	3.90900
1	8	51200	0.00618	0.00429	0.00444	0.00406
3	8	51200	0.71679	0.35707	0.18361	0.12534
7	8	51200	9.67034	4.82574	2.45543	1.64566
13	8	51200	50.5352	25.1750	12.8196	8.65828
1	16	102400	0.01108	0.00804	0.00676	0.00638
3	16	102400	1.49845	0.76739	0.37532	0.25248
7	16	102400	21.7944	10.8645	5.43781	3.64330
13	16	102400	123.317	61.4237	30.9508	20.5567
1	32	204800	0.02002	0.01422	0.01193	0.01063
3	32	204800	3.02729	1.51200	0.76098	0.51041
7	32	204800	45.6608	22.7674	11.3818	7.62043
13	32	204800	262.660	130.975	65.7338	43.6484
1	36	230400	0.07806	0.01596	0.01324	0.01196
3	36	230400	3.66452	1.67630	0.85550	0.56357
7	36	230400	50.0662	24.7522	12.5119	8.24016
13	36	230400	287.257	142.382	70.8652	47.2883

Appendix F: OpenCL GPU implementation mean running time performance data table.

Kernel Size	Slices Per Chunk	Voxels	Radeon 6850
1	1	6400	0.00250
3	1	6400	0.00317
7	1	6400	0.00721
13	1	6400	0.03045
1	2	12800	0.00226
3	2	12800	0.00365
7	2	12800	0.01154
13	2	12800	0.05827
1	4	25600	0.00258
3	4	25600	0.00442
7	4	25600	0.02453
13	4	25600	0.12450
1	8	51200	0.00303
3	8	51200	0.00681
7	8	51200	0.06046
13	8	51200	0.28417
1	16	102400	0.00449
3	16	102400	0.01125
7	16	102400	0.10882
13	16	102400	0.68811
1	32	204800	0.00735
3	32	204800	0.02019
7	32	204800	0.21176
13	32	204800	1.34306
1	36	230400	0.00764
3	36	230400	0.02197
7	36	230400	0.23935
13	36	230400	1.51343

Appendix G: FPGA sinc interpolation mean running time raw data table for static and dynamic parallel sinc kernels data table.

Kernel Size	Slices Per Chunk	Voxels	Dynamic Kernel	Static Kernel
1	1	6400	0.030	N/A
3	1	6400	0.009	N/A
7	1	6400	0.049	N/A
13	1	6400	0.283	0.293
1	2	12800	0.008	N/A
3	2	12800	0.013	N/A
7	2	12800	0.100	N/A
13	2	12800	0.579	0.552
1	4	25600	0.008	N/A
3	4	25600	0.024	N/A
7	4	25600	0.219	N/A
13	4	25600	1.239	1.175
1	8	51200	0.011	N/A
3	8	51200	0.044	N/A
7	8	51200	0.524	N/A
13	8	51200	2.809	2.661
1	16	102400	0.014	N/A
3	16	102400	0.086	N/A
7	16	102400	1.173	N/A
13	16	102400	6.841	6.472
1	32	204800	0.023	N/A
3	32	204800	0.168	N/A
7	32	204800	2.451	N/A
13	32	204800	14.56	13.78
1	36	230400	0.025	N/A
3	36	230400	0.192	N/A
7	36	230400	2.670	N/A
13	36	230400	15.80	14.94

Curriculum Vitae

Name:	Andrew Kope
Post-secondary Education and Degrees:	University of Western Ontario London, Ontario, Canada 2007-2011 Honors B.A. University of Western Ontario London, Ontario, Canada 2011-2012 Post Degree Module
Honours and Awards:	NSERC Undergraduate Summer Research Award 2011
Related Work Experience	Teaching Assistant The University of Western Ontario 2012-2013

Publications:

Kope, Andrew B, Rose, Caroline C, Katchabaw, Michael. (2013). Modeling Autobiographical Memory for Believable Agents. Proceedings of the 9th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.