Accelerating XD-GRASP MR Image Reconstruction

Author:
Thomas Yung
ty516

CID:
01203300

Supervisors:
Prof. Wayne Luk
Dr. Andreas Wetscherek
Marco Barbone

Submitted in partial fulfillment of the requirements for the MEng degree in Computing of Imperial College London

Dedicated to my Grandfather, Brian Yung.
University taught me how to be good with computers,
but you taught me how to be a good man.

Abstract

There is a large time delay between the acquisition of MRI scans and images being reconstructed, during which the anatomy of the patient may change. Given its clinical importance, there have been efforts to make this process faster. Specialised hardware can be used to accelerate XD-GRASP image reconstruction of MRI scan data, by taking advantage of the fine-grained optimisations possible. This project introduces a CPU, GPU and FPGA implementation of the XD-GRASP algorithm to produce 4D MR images and explores what the hardware requirements are for real-time 4D MRI reconstruction. Identifying and resolving bottlenecks and optimisations to the main body of the algorithm achieve a 4.05x speed-up over the original MATLAB implementation and a 1.54x speed-up over a baseline C++ implementation executing homogeneously on a CPU. Further hardware (GPU) is used to accelerate the largest computational burden of the algorithm, the non-uniform Fast Fourier Transform, to yield a 1.88x performance increase over the C++ implementation and a reduction of the MATLAB XD-GRASP execution time from 65.118s to 9.811s, a 6.63x speed-up.

Acknowledgments

I give my thanks to my supervisors Wayne Luk (ICL) and Andreas Wetscherek (ICR) for their support and guidance through the project. Their knowledge and expertise has been invaluable.

I reserve special thanks to Marco Barbone (Maxeler) for his assistance, advice and suggestions from the start and for committing his time to ensuring the project could be successful.

I would also like to thank my close friends and family for their continuous support and belief in me. At times it felt like you carried me through.

I thank the Mathematics Department at Urmston Grammar School who gave up their unpaid time to teach me Further Maths and allow me to pursue my career despite changing my path so late. Without them, none of this would have been possible.

Lastly, I thank my sister, Louisa, and Nana, Diane, who have stuck by me through the best and worst of times and are the people I treasure most in my life.

Contents

1 Introduction

Britons once voted cancer as being the most feared disease [1], with statistics showing that ‘every 2 minutes’ a person in the UK is diagnosed with a form of cancer [2]. The need for a fast, efficient way of treating cancer has never been more important.

In radiotherapy, a cancer treatment in which doses of radiation are targeted at an area of cancerous cells in order to kill them, physicists and dosimetrists plan a radiation dose using a three-dimensional image of the patient’s interior, which reveals the size, shape and location of the tumour. However, during the delay between scanning the patient and executing the plan, which is typically in the order of weeks, the tumour may grow and the anatomy of the patient may change (due to weight loss), causing doses to be given on the side of safety – they are weaker and, consequently, multiple are given throughout a therapy course.

In order to address the problems presented by the time delay, adaptive radiotherapy aims to provide a real-time dose recommendation requiring calculations, in this case the image reconstruction, to complete fast enough for the suggestion to be applied whilst the patient is in treatment. Henceforth, the objective of this project is to accelerate MR image reconstruction and to learn whether it is possible for MRI reconstruction to execute fast enough for adaptive radiotherapy.

For the patient, alongside speed, the comfort of the treatment is also important. Physical movement in the patient, from breathing, causes reconstructed MR images to be blurred when a time-indifferent reconstruction algorithm is used and, whilst the patient could be instructed to hold their breath, this is not always a practical or possible solution. In addition, a scan taken during a breath-hold produces images only valid for that specific breath-hold, meaning the patient would have to be instructed to reproduce the same breath-hold during the treatment, for the images were to be used reliably – a further impracticality. Physiological motion can also cause variation in the location of the tumour, motivating the need for MR images with temporal resolution (an additional dimension to 3D images). XD-GRASP [3] offers the ability to reconstruct 4D MRI scan images from data obtained under free-breathing, which allows the patient to be as comfortable as possible during an emotionally stressful process, making the algorithm a suitable choice for optimisation.

Thus, this project accelerates XD-GRASP by using an FPGA and a GPU in order to provide the best treatment possible to the patient. To achieve this, this report outlines the following work:

- Section 3: Design Flow and Performance Modelling
  - A software model was implemented to provide a baseline performance and framework for optimisation validity-checking
  - A performance model was produced to identify bottlenecks and anticipate the effects of optimisations
- Section 4: Optimisation
  - Bottlenecks are mitigated by maximising parallelism and hardware utilisation
- Section 5.1: Implementation
  - An FPGA is used to accelerate XD-GRASP
  - Additional hardware (a GPU) is added to explore further acceleration
- Section 6: Performance Evaluation
  - The limits of the used hardware are assessed for 4D MRI reconstruction
  - Hardware requirements for real-time 4D MRI reconstruction are discussed

XD-GRASP brings new difficulties to MRI reconstruction as the extra dimension, time, multiplies the effort required to produce a complete reconstruction. In order to qualify as real-time reconstruction, the extra dimension would have to be handled such that the algorithm execution time is comparable to time-indifferent reconstruction algorithms (see Section 2.7). Additionally, the reconstruction for each respiratory phase is not isolated from others, ruling out segmented parallel reconstructions for each phase. A temporal variation operator is used frequently, making use of data from multiple phases.

Preliminary tests show a reconstruction of a 4D image comprised of 320x320 2D images, with a depth of 147 and 8 time steps, takes 9577 seconds (equivalently around 159 minutes or 2.66 hours). This time is an approximation to the total running time of the existing XD-GRASP implementation, as the algorithm make take longer or shorter depending on the image data and whether or not it requires a larger number of iterations (determined by the algorithm) to construct clear images. Acceleration of this algorithm should aim to achieve a similar reconstruction in the order of minutes, to be used in adaptive radiotherapy.

Ultimately, producing a system capable of executing XD-GRASP in satisfactory time (approaching real-time) for adaptive radiotherapy is the goal. This project aims for iteratively reconstructed, respiratory-resolved (4D) MR images in the order of seconds. Once achieved, adaptive radiotherapy may allow cancer treatment to be completed during fewer treatment sessions and thus reduce the turmoil endured by the patient.

2 Background

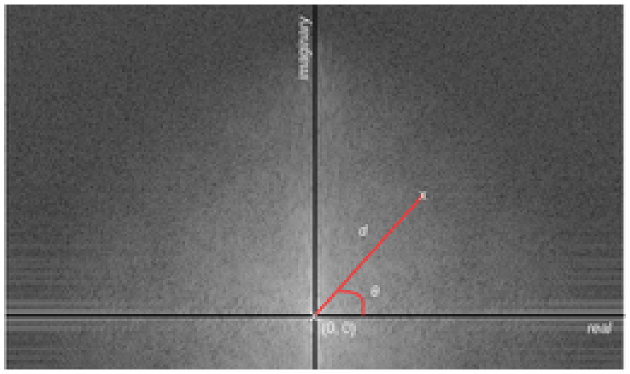
This section introduces the key concepts and algorithms used in present day MRI data processing. It begins by outlining the basic principals behind MRI data collection and transformation to images usable by doctors, physicists and dosimetrists. Following this, detail is provided on the underlying mathematical concepts which help achieve such transformations. The algorithm to be accelerated, Golden-angle Radial MRI with Reconstruction of Extra Motion-State Dimensions Using Compressed Sensing (XD-GRASP) [3], is outlined along with the basic components of FPGAs and the performance benefits they bring.

2.1 MRI

The invention of magnetic resonance imaging (MRI) allowed doctors to perform non-invasive observations of their patients' internal organs and tissues [4]. The scanner performs the scan (the mechanics of which go beyond the scope of this project) to produce data that can be reconstructed into a 3D image.

An MRI scanner detects signal from hydrogen atoms bound in biological tissue to obtain a signal [5]. This signal is then measured using receiver coils and reflects the interior of the patient.

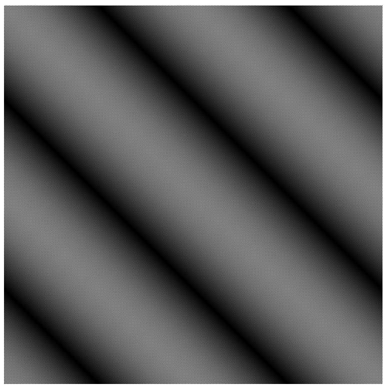
The data obtained from MRI scanners is obtained in the frequency domain referred to as k-space. A location in k-space is represented by its distance from the origin and angle (anticlockwise from the positive real-axis). The intensity found at a location in k-space data reflects the corresponding wave's (in image space) contribution to the image [6] (see Figure 1).



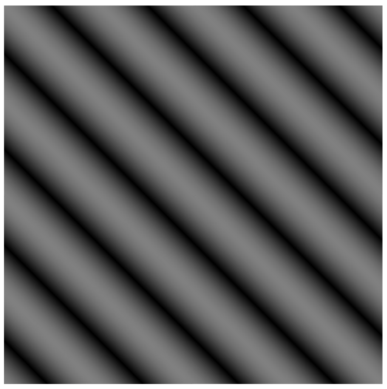
(a)

Figure 1: A location in k-space with distance d from the origin and angle θ.

The correspondence between k-space and image space is best illustrated by considering a pair of related locations in k-space. A given location in k-space corresponds to a wave in image space, as seen in Figure 2. The angle of the point in k-space reflects the direction of the wave in image space and the distance of a k-space location from the origin reflects the frequency of the wave in image space. Comparing a location (Figure 2, (a)) with another k-space location (Figure 2, (b)) at equal angle but larger distance (from the origin), the direction of the wave is the same however the periodicity is smaller (or equivalently the frequency is higher).



(a)



(b)

Figure 2: Image space realisations of one data location with amplitude d from the origin (a) and with amplitude $2d$ from the origin (b).

The intensities of the k-space locations are used to perform a weighted sum of the waves, producing an image, as can be seen in Figure 3.

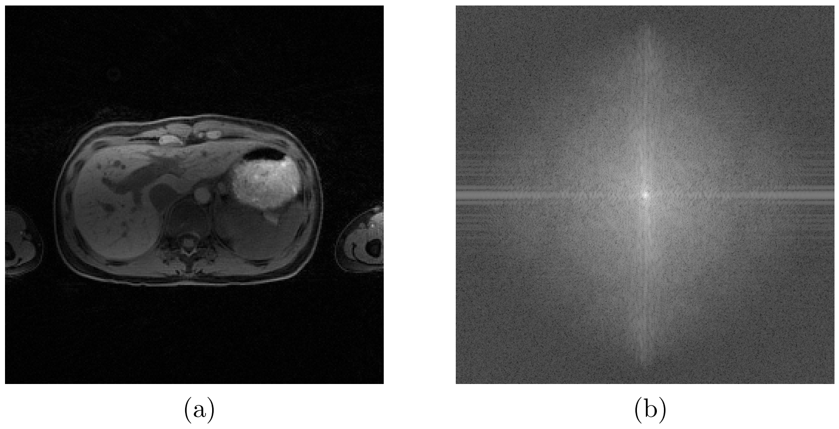


Figure 3: Corresponding representations of the same MRI data in image space (a) and k-space (b).

2.2 Fourier Transform

As described, there is a correspondence between k-space (the data the MRI scanner measures) and image space (human process-able images). In order to translate between the two, Fourier Transform is used:

$$S(\vec{k}) = \int_r M(r)B_1(r)e^{-2\pi(\vec{k}\cdot\vec{r})} \tag{1}$$

Where r denotes the image space, k denotes the k-space, M represents the image and B_1 represents the coil sensitivity.

Joseph Fourier (1768 - 1830) famously observed that a complex signal can be decomposed into a sum of plain sinusoidal waves [7] (a Fourier series) and since this discovery, initially applied to the solving of differential equations, has been utilised in a broad range of applications such as sound analysis and medical imaging.

Fourier Transforms has many forms (types) and uses which are beyond the scope of this project but are used in XD-GRASP. In optimising the algorithm, the original mathematics of it do not change and thus it is non-essential to cover them here. This section gives an overview of the libraries used to execute non-uniform Fast Fourier Transform, a variant of the aforementioned Fourier Transform.

2.2.1 Non-Uniform Fast Fourier Transform (NUFFT)

In the case when the data is not uniformly sampled (in either or both the time or frequency domain), the ordinary discrete Fourier Transform cannot be used. In MRI, in order to shorten the acquisition time of MRI signals, data are undersampled. Non-Cartesian sampling has the advantageous property of incoherent undersampling artefacts, which is exploited in compressed sensing reconstructions [8]. However, it requires an alternative approach to Fourier Transforms: the non-uniform Fast Fourier Transform [9].

FFTW [10], a state-of-the-art C++ library for Fast Fourier Transform, aims to perform the operation faster by allowing a pre-transform planning step which aims to build a more efficient plan to calculate the Fourier Transform, taking into account the input size and contents. In addition, the principals of Fast Fourier Transform have been applied to the non-uniform case too, with the FINUFFT library [11] being the C++ tool used for this (which in turn calls the FFTW library). This library evaluates the following equations, in the 2D case [12], whilst bringing greater precision and speed over a naive implementation. Here $x[j]$ and $y[j]$ are the k-space locations for sample j, k_1 and k_2 are image space locations and $c[j]$ is the source strength of sample j:

- Type 1:

$$f[k_1, k_2] = \sum_{j=0}^{N-1} c[j] \cdot e^{k_1 \cdot x[j] \ + \ k_2 \cdot y[j]} \tag{2}$$

- Type 2:

$$c[j] = \sum_{k_1, k_2} f[k_1, k_2] \cdot e^{-(k_1 \cdot x[j] \ + \ k_2 \cdot y[j])} \tag{3}$$

As the computation of the NUFFT is on a high volume of floating-point numbers, it lends itself to being performed on graphical processing units (GPUs) which have specialised hardware for these calculations. Thus came the creation of the gpuNUFFT library [13], which utilises NVidia's CUDA [14] to produce a NUFFT implementation taking advantage of GPUs.

2.3 Supporting Mathematical Processes

This section introduces mathematical processes which will be of use and describes their role in XD-GRASP.

2.3.1 L2-Norm

The L2-Norm [15] provides a reduction of a complex vector x to a single value based upon the norms of its coefficients:

$$||x||_2 = \sqrt{\sum_{k=1}^{n} |x_k|^2} \tag{4}$$

Where (for complex number $x_k = a + b\mathrm{i}$):

$$|x_k| = \sqrt{a^2 + b^2}$$

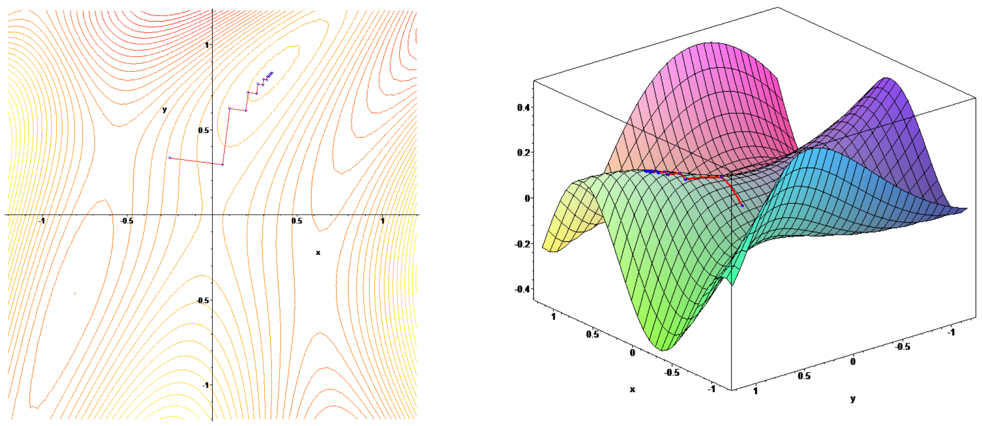
The L2-Norm is used in XD-GRASP for the objective function during the conjugate gradient method (Section 2.3.3) and in the decision to early-return from the conjugate gradient method.

2.3.2 Gradient Descent

Before covering the conjugate gradient method, it is useful to first introduce gradient descent, to illustrate the broad mechanics of the two algorithms.

Cauchy (1789 – 1857) proposed an algorithm for finding a local minimum (local to an initial guess) of a function [16]. By computing the gradient for a set of parameters, the values of these parameters can be shifted in the direction which leads to the greatest decrease in the function's result. For example, in two-dimensional space (axes x and y) with a function, f, giving the height of terrain at its input coordinates ($f(x, y) = z$), gradient descent finds a local minimum from an initial input point (see Figure 4).

In order to limit the running time of this algorithm, a gradient threshold is set such that descent terminates if the absolute value of the maximum negative gradient is lower than the threshold. Alternatively, only a predefined number of iterations may be allowed.



(a) Contour view (b) Surface view

Figure 4: Gradient descent performed with 12 iterations. (Image ref: [17])

When the initial guess is in a region with relatively neutral gradient (for example as in Figure 4), gradient descent is slow to converge on a minimum. However, since this drawback is data-dependent, it is not possible to generalise about the number of iterations to limit the algorithm to. In addition, gradient descent does not indicate the magnitude of a parameters shift – only in which direction. For this reason gradient descent is used in conjunction with other algorithms to determine the magnitude of the parameters update (see Section 2.3.4).

2.3.3 Conjugate Gradient Method

One weakness seen in gradient descent is that the zig-zag path it takes to converge on a local minimum is slow and can take many more interactions than the n dimensions of the data. An alternate approach, the conjugate gradient method [18], overcomes this by searching n directions such that progress towards the minimum in one direction does not affect progress in any of the other directions.

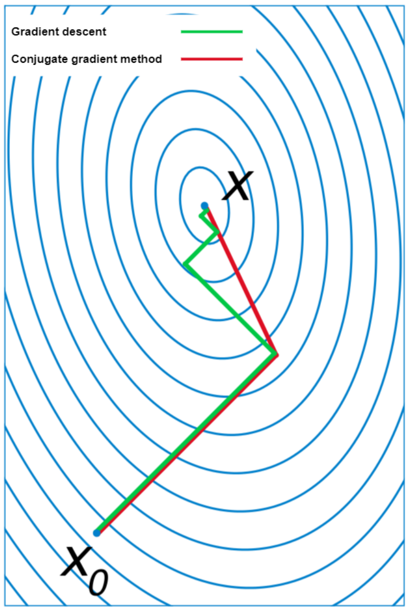


Figure 5: The conjugate gradient method (red) is more direct than gradient descent (green) in finding a minimum.

The conjugate gradient method is used in XD-GRASP to perform iterative improvements to the reconstruction.

2.3.4 Backtracking Line Search

Backtracking line search is an algorithm used to determine the magnitude of a step in a given direction (changing inputs) in order to minimise a function [19]. It refines an initial step size estimate by backtracking/reducing it until the function output decreases by at least some threshold amount which is computed using the local gradient of the function (the Armijo-Goldstein condition [20]).

Essentially, the search is performed by reducing an initial step size by a factor of τ (a search parameter) until the Armijo-Goldstein condition is fulfilled. This means the performance of the algorithm can be poor when τ is not chosen well. A step size is considered adequate when, for function f, initial position x, direction p, step size α and further search parameter c, the Armijo-Goldstein condition is satisfied:

$$f(x + \alpha p) \leq f(x) + \alpha c m \tag{5}$$

Where:

$$m = \nabla f(x)^T p$$

The key benefit that backtracking line search brings is alleviating the load on computational resources. When searching for the optimal parameters with respect to a cost function, backtracking line search avoids a large number of small steps by beginning with larger steps and then reducing the step size. The cost function for this project is given in Section 2.4, Equation 6, as the task at hand is posed as an optimisation problem.

As with gradient descent and the non-linear conjugate gradient method, this algorithm may be terminated early by constraining it to a maximum number of iterations.

2.4 XD-GRASP

Feng et al. proposed the use golden-angle radial sampling, compressed sensing (GRASP) [21] (a technique for reconstructing a signal from fewer samples), and parallel imaging[1] to produce an algorithm capable of processing an MRI signal into an image under free-breathing.

Building on the algorithm proposed by iGRASP [23], Feng et al. refined their approach further by exploiting sparsity along other *eXtra Dimensions* (XD), such as the total difference/variation between different phases of the respiratory cycle [3].

Data is obtained as radial samples/spokes, a vector along which measurements are taken, on a stream of k-space data in three dimensions (2D radial samples are stacked at even intervals in the z-dimension). Consequently each spoke corresponds to a different time. In order to group these spokes by the stage in the respiratory cycle in which they were taken, XD-GRASP includes a pre-processing step, in which a respiratory signal is obtained from the k-space data observed.

As per the golden-angle sampling scheme, the spokes intersect where $k_x = k_y = 0$ (k-space coordinates) – this point is the central profile of the spoke (Figure 6. The central profiles are taken for every spoke at each z-dimension interval and used to determine the respiratory motion signal using principal component analysis [24].

[1]Using multiple receiver coils in the MRI scan [22].

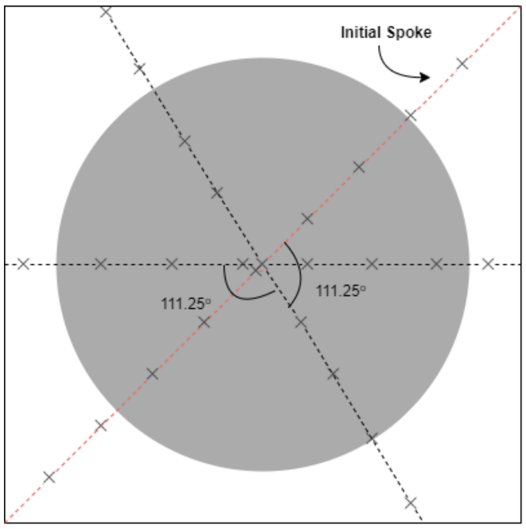


Figure 6: Golden-angle sampling of a 2D area. The placement of spokes depend on the the previous spoke, with the new orientation of the spoke being that of the previous spoke plus 111.25°. Samples are taken along a spoke.

With this signal, a sample spoke can be categorised by the stage of respiratory motion it was taken at, thus grouping the data. Here, sorting is performed according to the phase of the respiratory cycle instead of the time of acquisition. As seen in Figure 7, the goal is to group the spokes that were sampled in the same respiratory phase – denoted by the coloured dashed boxes.

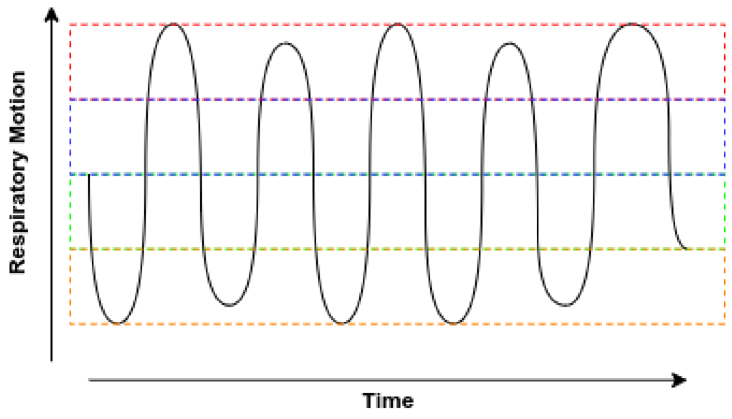


Figure 7: Respiratory motion over time with the amplitude gating denoted by the dashed boxes.

XD-GRASP then aims to find d which minimises the following:

$$d = \text{arg min}\Big\{||F \cdot C \cdot d - m||_2^2 + \lambda||S \cdot R \cdot d||_1\Big\} \tag{6}$$

Where d is the image series (one image reconstructed from each temporal frame), S is the sparsifying transform applied along the extra respiratory-state, m is the k-space data, C is the coil-sensitivities and F is the non-uniform Fourier Transform operator (Section 2.2.1). λ is a search parameter which controls the weighting given to the sparsifying transform.

As before, the conjugate gradient method (Section 2.3.3) is used to achieve this minimisation.

Note that in the paper [3], the minimisation function stated is for the cardiac imaging use case and thus includes a further sparsifying transform in the cardiac motion dimension, which is not the focus of this project and therefore omitted.

2.5 Image Similarity Measures

It is useful to compare two images statistically when evaluating the performance of this project. Detailed below are the two most popular metrics [25].

2.5.1 Mean Squared Error

By linearly ordering the pixels of an image, of which there are p, the squared pixel intensity differences between corresponding pixels in images a and b can be summed to produce a single value (note: the ith pixel of image a is denoted by a_i):

$$MSE = \frac{1}{p}\sum_{i=0}^{p}(a_i - b_i)^2 \tag{7}$$

Further, it is possible to normalise the mean squared error by dividing by the square of the range of values a pixel can take, R:

$$nMSE = \frac{1}{R^2}\, MSE \tag{8}$$

Whilst simple and intuitive, it has been shown to be misleading [26] and can be shown to give more positive results for images with poor quality.

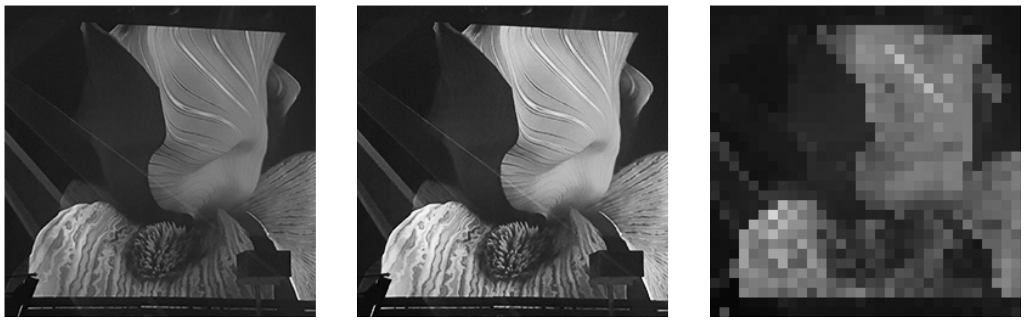

(a) nMSE = 0 (b) nMSE = 0.0151 (c) nMSE = 0.0056

Figure 8: Normalised mean squared error, rounded to four decimal places, ranks image (c) (a pixelated version of the reference image (a)) above (b) (a brighter version of the reference image).

2.5.2 Structural Similarly

Zhou Wang et al. [27] address the weaknesses of mean squared error with a quality index Q which combines the correlation, mean luminance (pixel intensity) and image contrast to produce a score in the interval $[-1, 1]$:

$$Q = \frac{\sigma_{ab}}{\sigma_a \sigma_b} \cdot \frac{2\bar{a}\bar{b}}{(\bar{a})^2 + (\bar{b})^2} \cdot \frac{2\sigma_a \sigma_b}{\sigma_a^2 + \sigma_b^2} \tag{9}$$

$$= \frac{4\sigma_{ab}\bar{x}\bar{b}}{(\sigma_a^2 + \sigma_b^2)\left[(\bar{a})^2 + (\bar{b})^2\right]} \tag{10}$$

Where:

$$\bar{a} = \frac{1}{N}\sum_{i=1}^{N} a_i, \quad \bar{b} = \frac{1}{N}\sum_{i=1}^{N} b_i$$

$$\sigma_a^2 = \frac{1}{N-1}\sum_{i=1}^{N}(a_i - \bar{a})^2, \quad \sigma_b^2 = \frac{1}{N-1}\sum_{i=1}^{N}(b_i - \bar{b})^2$$

$$\sigma_{ab} = \frac{1}{N-1}\sum_{i=1}^{N}(a_i - \bar{x})(b_i - \bar{y})$$

In images the output of this quality index (and likewise the relationship between pixels) is space variant – pixels in a similar region are more likely to be similar. Thus in order to apply this metric to an image, rather than processing the whole image in one calculation, it is more appropriate to obtain local quality indexes through a sliding window, producing M local quality indexes. The overall quality index is an average of the local indexes:

$$Q = \frac{1}{M} \sum_{j=1}^{M} Q_j \tag{11}$$

Wang's findings showed the quality index ordering of distorted images was equivalent with the mean ranking of the same images when shown, alongside the original, to human subjects. Whilst the subject ordering is subjective, it displays that the quality index can characterise the features humans consider when analysing images.

For the same images as tested for the mean squared error, the results for SSIM accurately reflect the similarity between the images:

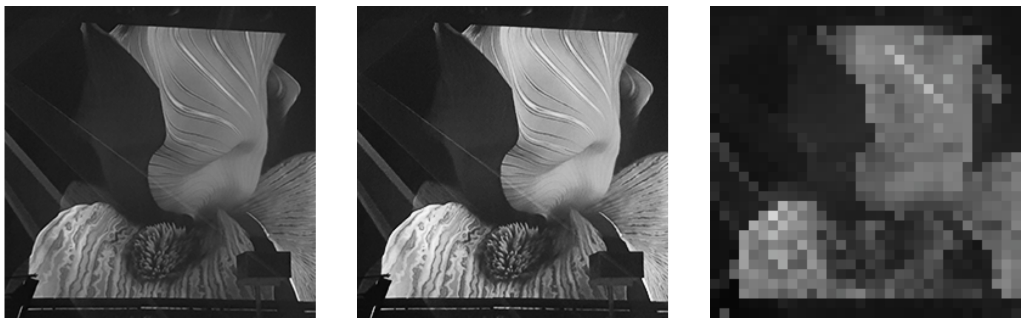

(a) SSIM = 1 (b) SSIM = 0.92 (c) SSIM = 0.57

Figure 9: Structural similarity ranks image (b) above (c) when either is compared to the reference image (a).

2.6 Field Programmable Gate Arrays

This section explains the key concepts of field programmable gate arrays (FPGAs) and discuss the Xilinx® VU9P [28], the specific board in use for this project, to implement custom accelerators for XD-GRASP. This detailed take on optimisation yields performance improvements up to several order of magnitude.

However this speed-up comes at a cost. Synthesising an algorithm on an FPGA takes a long time, often days, and the platform does not lend itself to easy debugging. To this end, Maxeler [29] provide MaxCompiler, a tool which abstracts hardware details into a high-level description language and further, a simulator to allow the validity of hardware without the need for synthesis.

In this project an FPGA is used to accelerate the non-NUFFT parts of the algorithm, as it allows a custom architecture to be designed which can take advantage of parallelism in the algorithm design [30], something less possible on general-purpose hardware such as CPUs and GPUs.

2.6.1 Components

There are four key hardware components on an FPGA, according to Maxeler terminology.

- **LUT (Lookup Table)** Rather than having physical logic gates on the FPGA, it instead saves a lookup table reflecting the outputs for given inputs. This table is a truth table for three inputs meaning any circuit for three inputs and one output can be represented.

- **DSP** DSPs are the hardware for multiplication. They are rectangular multipliers, accepting two inputs of size 18 bits and 27 bits on the VU9P, with pre- and post-adders, to allow additions with no further computational burden. If multiplication for higher dimensions is required, DSPs are used in combination with one another.

- **Flip Flop** Flip flops are circuitry implementations of registers, capable of storing values.

- **B/URAM** BRAM and URAM are the on-board memory (equivalent to the cache of a CPU)

In addition to on-board memory there is external memory, DRAM, which is slower to access (equivalent to RAM in conventional computers). Sequential accesses to on-board memory perform considerably better than random access. The FPGA in use, the VU9P, is on a MAX5 board with DRAM mounted on the PCB (printed circuit board).

2.6.2 Optimisation Targets

When optimising an algorithm for an FPGA, there are two main bottlenecks to consider: communication and computation.

Accesses to on-board memory are fast, regardless of sequential or random locations. However, accesses to DRAM are relatively slow. The memory controller generated by MaxCompiler defines a unit of access to DRAM, referred to as the 'burst size'.

In accessing DRAM, attention must be given to the size of data being accessed (preferably utilising a whole burst at a time) and access pattern (sequential vs random). Should fewer bits than a burst be required, a full burst is read and the excess bits are discarded, an inefficiency.

In order to run efficiently, the algorithm should aim to maximise the utilised hardware resources. In the case of DSPs, for example, if there are 100 DSPs on the board, it is possible to achieve 100 multiplications in one clock cycle.

2.6.3 Optimisation Strategies

There are a number of strategies that can be employed in order to optimise an algorithm on an FPGA.

- **Pipelining** In order to increase the throughput of the board (time between two inputs), pipelining is used. The algorithm is split into disjoint stages and data flows through them. Once a portion of data has completed the initial stage, regardless of its progress through the algorithm as a whole, a second portion can begin processing in the initial stage. This strategy is also applicable as a programming paradigm, in order to best utilise resources.

- **Double Buffering** During the time in which one data chunk is being used in the computation, a second chunk can be loaded from DRAM so that it is ready to be processed immediately after the first chunk has completed. This idea can be applied to the output of an algorithm whereby the output is written to DRAM as the algorithm computes a different output. This strategy allows the time costs of communication and computation to overlap, such that execution time, t, is the maximum of the communication and computation time ($t = \max(t_{compute}, t_{transfer})$) instead of the sum of the two ($t = t_{compute} + t_{transfer}$).

2.6.4 Xilinx® VU9P

For this project, the architecture is implemented using MaxCompiler version 2019.2 and Vivado 2018.3. The FPGA device targeted is a Maxeler's MAX5C Dataflow Engine (DFE). The MAX5CDFE is based on the Xilinx VU9P 14nm/16nm FinFET FPGA, consisting of:

- LUTs: 1,182,240
- FFs: 2,364,480
- BRAMs: 2,160
- DSPs: 6,840

2.7 Related Work

Previous studies have shown how FPGAs can be utilised for MRI reconstruction in real-time, with [31] demonstrating reconstruction of a 256x256 pixel image in 0.164ms using an implementation of SENSE [32] on a Xilinx® Virtex-6 ML605 board operating at 200MHz. Further work [33] enables the reconstruction of 128x128 images at 400 frames per second (equivalently 0.0025s per reconstruction) on a XC6SLX45 chip operating at 40MHz.

Alternatively, GPU implementations of 3D MRI reconstruction exist with [34] achieving a 128x128x128 reconstruction in 49 seconds using an NVIDIA® Quadro FX 5600.

Normalising these reconstruction times by the number of pixels allows the following comparison, Table 1, to be drawn. Whilst this project implements a 4D reconstruction algorithm, benchmarks were produced for 8 2D images, one for each respiratory phase, each of size 320x320. This makes the effective dimension, for comparison purposes, 3D (width, height, time).

Impl.	Hardware	Total Recon. Time (s)	Dim.	Number of Pixels	Time per Pixel (s)
[31]	FPGA	0.164	2D	65536	0.000002502
[33]	FPGA	0.0025	2D	16384	0.000000153
[34]	GPU	49.0	3D	2097152	0.000023365
This project	CPU/GPU /FPGA	9.7836	3D	819200	0.000011943

Table 1: A comparison of MRI reconstruction implementations.

Whilst this work could be used for adaptive radiotherapy, the reconstruction algorithms implemented do not support radial sampling schemes, such as that used in XD-GRASP, and further have limited temporal resolution.

3 Design Flow and Performance Modelling

Before optimisations can be made, it is important to identify the areas of the algorithm which consume the most time and thus give an indication of the bottlenecks. To this end, a software model and a performance model were produced to provide a framework for strategies to be evaluated upon, assisting the planning phase of this project. The model also maximises the speed-up whilst minimising the code-written to achieve said speed-ups and is parametric making it easily extended to other (even future) platforms.

3.1 Design Flow

In order to design the system using a CPU, FPGA and GPU, the following design flow was used:

- MATLAB Implementation: The XD-GRASP demo [35] from the original paper serves as a reference for image accuracy.
- Software Model: A C++ implementation provides a performance baseline and, further, a platform to trial optimisations to ensure image accuracy does not degrade.
- Performance Model: This captures the timing characteristics of the XD-GRASP algorithm implementation on the FPGA, CPU and GPU and therefore highlights the bottlenecks. Further, the total resource utilisation can be checked to ensure the design fits onto the FPGA.
- Optimise: Using the software model to check validity, the bottlenecks highlighted by the performance model can be addressed using the optimisations described in Section 4.
- Simulating the FPGA Behaviour: The software model is modified to simulate the FPGA behaviour, including the communication between the CPU and the FPGA. This may expose further optimisations to be made.
- Predict the FPGA Performance: Using the performance model, the performance of the FPGA implementation can be predicted and the hardware resources can be prioritised for bottlenecks.

- Implement on the FPGA: The optimised design can be implemented in MaxJ (Maxeler's high-level description language) and synthesised onto the FPGA.

3.2 XD-GRASP

Excluding the pre-processing step of the algorithm, which sorts the data into respiratory phases, the following steps of XD-GRASP were to be accelerated:

- Compute an initial reconstruction, for each respiratory phase
- Perform the conjugate gradient method to make iterative improvements to the reconstructions (see Section 2.3.3)
  - Compute the gradient of the reconstruction
  - Perform backtracking line search (see Section 2.3.4)
  - Apply the gradient to the reconstruction, using the step size found in backtracking line search
  - Iterate the previous steps until a maximum number of iterations or the L2Norm (see Section 2.3.1) of the gradient is lower than some threshold (i.e. the improvements made to the image are not significant enough)

3.3 Software Model

A C++ implementation of the XD-GRASP demo code [35] was implemented, providing the aforementioned framework to test the validity and performance of optimisations to be tested upon. This implementation also acted as a preliminary optimisation in-itself as the lower-level control offered by C++ over the original abstracted (and interpreted) MATLAB implementation allows for better use of computation resources (see Section 6 for results and discussion).

The C++ implementation was optimised to provide a reliable baseline for the accelerated implementation to be compared to.

3.4 Modelling the Algorithm

The performance model reflects the dataflow of the algorithm through parametric equations and the orchestration of parallel computation on the hardware used.

With an initial C++ implementation, an initial performance model could be produced to model the algorithm steps (steps which would be ported onto an FPGA). This brought the ability to observe the time cost for different stages of the algorithm and therefore highlight those which held back the execution time the most. Once these bottlenecks were identified, the algorithm design was changed, with validation from a modified software model, to mitigate the bottlenecks.

The optimisations made to the algorithm, for acceleration on an FPGA, do not necessarily accelerate a CPU implementation. Reducing the number of multiplications does not map to the same performance increment on different platforms. Therefore, rather than implementing changes in the software model and measuring performance of the it executing, the performance model was used to predict the performance increase or decrease of changes to both the algorithm and parameters of execution (such as the data-type and hardware resources). The software model only checked the correctness of optimisations. Unlike a C++ implementation, the performance model could also anticipate how the hardware characteristics impact the execution time as the FPGA execution time is predictable and runs for a manually set, fixed number of clock cycles.

Following the model of a C++ implementation of XD-GRASP, the modelled FPGA kernels (Section 3.5) were incorporated along with their compute times. Thus the orchestration of the FPGA kernels and NUFFT had to be emulated in the performance model to arrive at an overall algorithm execution time estimate, which took into account concurrent execution of the CPU and FPGA (discussed further in Section 4).

When determining a predicted execution time for the algorithm, benchmarks for the NUFFT had to be measured for both the CPU and GPU implementations. These are discussed further in Section 6.1. These times highlighted the NUFFT as a large bottleneck, with a comparison of the total NUFFT time ($= \#invocations \times benchmark$) and the measured C++ implementation performance revealing that 64.96% of the total execution time is spent on the NUFFT.

3.4.1 Modelling Example

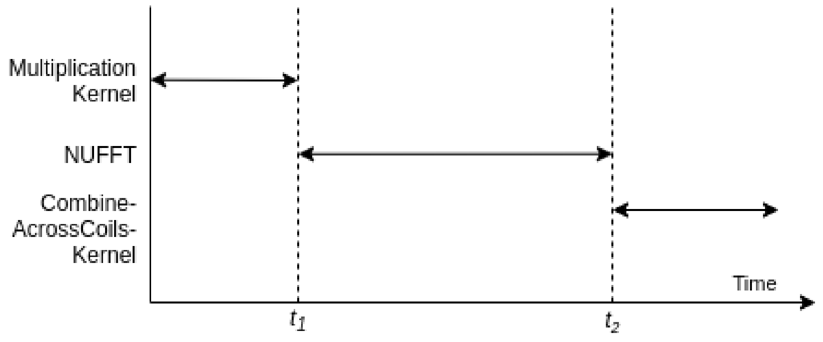
For example, to model the initial reconstruction, the following algorithm steps are considered:

- (FPGA: `MultiplicationKernel`) Element-wise multiply `kdatau` and `wu` (further detail on these datasets is given in Section 5.1) for every respiratory phase and coil
- (CPU/GPU) Perform a type 1 NUFFT on the output vector of each multiplication from the previous step
- (FPGA: `CombineAcrossCoilsKernel`) For each respiratory phase, combine the results of the NUFFT for each coil

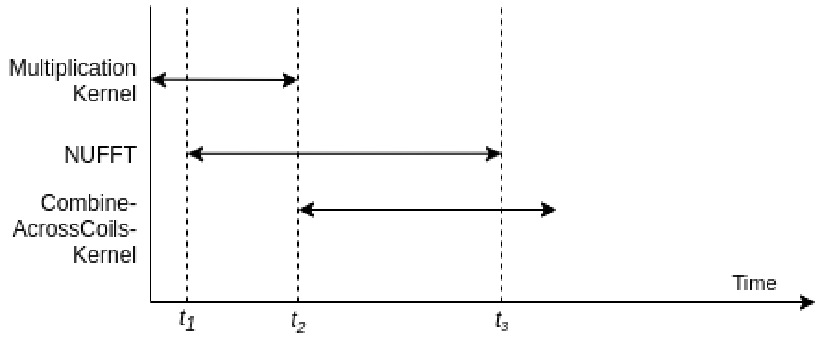
Naively, these steps would be performed sequentially in the performance model and the execution time would be the sum of the times taken to complete each step. As shown in Figure 10 (a), the NUFFT operations begin after all of the multiplication results are obtained (t_1) and the combining begins after the NUFFT is fully complete (t_2).

However, after optimisations outlined in Section 4, the computation is overlapped to increase the utilisation of the CPU and FPGA (see Figure 10 (b)). In this case, the NUFFT operations begin as soon as the first multiplication result is output, (t_1), and the remaining multiplication outputs can be computed during the time it takes the NUFFT operations to complete. The model can be used to check that, aside from the first invocation of the NUFFT, the input is always available for a NUFFT operation when it is ready to begin (i.e. immediately after the last NUFFT).

Following this, when all of the NUFFTs for a coil have been computed, the results can be combined on the FPGA, starting at (t_2), when the FPGA has no more multiplications to do – something which can occur during the NUFFT operations, so long as all the required data has been output by the NUFFT. The final combine-across-coils operation can only begin once the final NUFFT operation has completed, t_3.



(a) A naive implementation.



(b) A hardware utilisation optimised implementation.

Figure 10: A comparison of a naive and optimised implementation to compute the initial reconstruction.

3.5 Modelling the FPGA

With the CPU resources and XD-GRASP algorithm steps benchmarked, the FPGA can be analysed and modelled. For this, the algorithm, this time incorporating an FPGA, was modelled similarly to the C++ code, with its own C++ implementation to simulate the porting of functionality onto an FPGA.

Modelling an FPGA required more rigorous planning, with data transfer and orchestration incurring new costs. FPGAs require manual coordination of data in the on-chip memory, a level of control only partially possible on a CPU. As the kernels in use were fully pipelined (since no elements were re-input into the computation), the compute time of kernel could be estimated using:

$$T_{compute} \approx \frac{E}{C \cdot E_c} + Fill\ Time \tag{12}$$

Where E is the number of elements to be processed, E_c is the number of elements processed per clock cycle and C is the clock frequency, in Hertz. The fill time, the time taken for the FPGA pipeline to fill with data, is assumed to be negligible in this calculation. It remains constant for various input sizes, so becomes negligible as the input size grows.

Since, on an FPGA for this algorithm's workload, the number of DSPs is a more constraining resource than LUTs or FFs, the compute time equation, Equation 12, can be simplified. This calculation can be performed by considering the number of (effective) multiplications performed during a kernel execution and the number of DSPs allocated to the kernel (as this reflects how many multiplications can occur simultaneously in one clock cycle). Simplifying Equation 12 gives:

$$T_{compute} \approx \frac{\#Multiplications}{\#DSPs\ Allocated \times Clock\ Frequency} \tag{13}$$

This model setup provided insight into which kernels demanded the most time and hardware resources and further allowed the quick evaluation of FPGA configurations which allocated fewer or greater resources to particular kernels. Thus, in planning the design of the algorithm components on the FPGA, the bottlenecks could be mitigated by re-allocating hardware resources, in most cases DSPs and B/URAM, to them. The effectiveness of DSP allocation was enabled by the hardware replication optimisation outlined in Section 4.

The communication time required for kernel inputs and outputs to be transferred to and from the FPGA was modelled by dividing the size of the data by the PCIe 3 data rate (15700MB/s), as shown in Equation 14. After the introduction of explicit actions, explained in Section 4.3, this cost was omitted. The explicit actions API gave finer control, allowing, with an asynchronous call, the communication cost of one kernel to overlapped with the compute time of another kernel invocation. The introduction, of explicit actions meant the compute time shifted from being I/O bound to being compute bound.

$$T_{communicate} = \frac{Data\ Size}{Data\ Transfer\ Rate} = \frac{Data\ Size}{15700} \tag{14}$$

As the model is parametric, it can be used to anticipate the performance achieved with different hardware. The model characterises the influence that the FPGA's hardware components have on the kernel execution times and therefore can be used to predict the hardware required to achieve a particular performance.

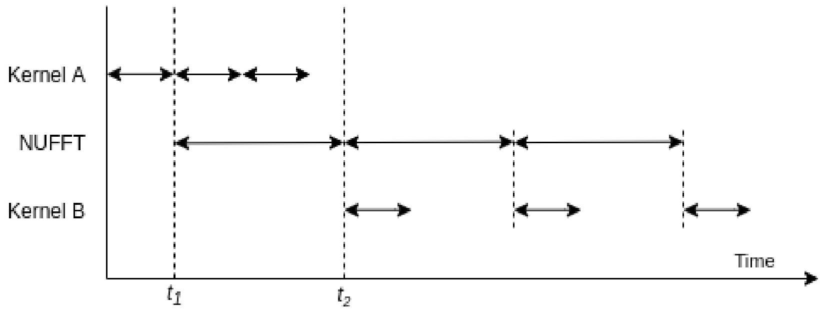
As with modelling the algorithm, modelling the FPGA usage was iterative, following the optimisations discussed in Section 4.

3.6 Model Limits

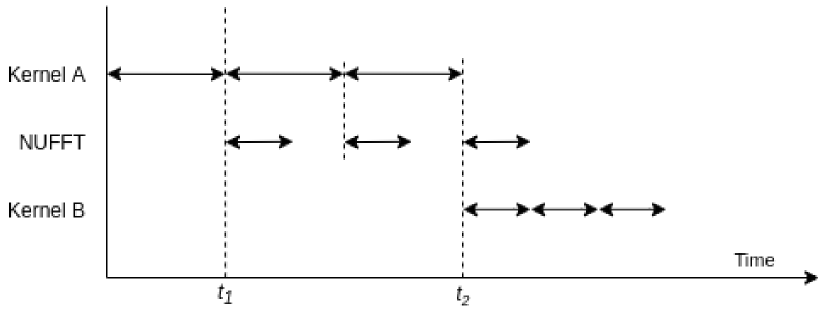
Analysis can be done to explore the limits of the performance model. As discussed in Section 4, most of the FPGA work is completed during the same time that the CPU or GPU is performing NUFFT. Thus, in order for the current model to be valid, in cases where simultaneous work occurs, the NUFFT must not *overtake* the currently planned FPGA work.

Consider Figure 11, in which a timeline of arbitrary overlapping CPU and FPGA work is presented, with arbitrary kernels A and B. Each NUFFT operation uses the output of a Kernel A operation and, likewise, each Kernel B operation requires the output of a NUFFT operation. Whilst the scale and kernels are not directly from the designed system, the concept and calculations appear in a similar form throughout the performance model.

In the first sub-figure, (a), the model calculates the time taken for the total computation by summing the time taken for one invocation of Kernel A, three NUFFT operations and one invocation of Kernel B. This is because the time taken to perform the NUFFT covers the time taken to execute the kernels, where both the NUFFT and FPGA can run simultaneously. On the other hand, if the NUFFT becomes shorter than the kernel, the same calculation no longer applies, as shown by the second sub-figure, (b).



(a)



(b)

Figure 11: In sub-figure (a), the NUFFT can only begin after the first output of Kernel A is produced, at t_1. Similarly, Kernel B can only begin after the first output of the NUFFT has completed, at t_2. Sub-figure (b) holds the same constraint for t_1 however Kernel B cannot begin until all of the tasks for Kernel A have completed (t_2), invalidating the a performance model used for (a).

By inspecting each case like this in the performance model, a bound for the maximum speed-up of the NUFFT that the model is still valid for can be calculated. To achieve this, consider Figure 11 (a). When speeding up the NUFFT, or equivalently shortening the arrow for it, there will come a point where the overall execution time is no longer equal to the time taken for one invocation of Kernel A, three NUFFT operations and one invocation of Kernel B. It is a this point that the performance model calculation is invalid.

Note that the CPU NUFFT and GPU NUFFT implementations of XD-GRASP both use the same calculations for overall execution time in the performance model. Therefore, the maximum speed-up of the NUFFT which the model is still valid for can be mapped to either case. The speed-up bound for the CPU NUFFT is: 136x. In other words, if the NUFFT operations accelerate by less than 136 times, the performance model will still be valid. The performance model can be used to identify cases where the time taken for kernels and the NUFFT is similar and thus highlight potential bottlenecks to be addressed once NUFFT no longer dominates time.

The modelling phase of the project draws focus to the slower parts of the algorithm which are constrained by memory or computation demands. Further it provided a platform for speculative improvements and optimisations to be made.

4 Optimisation

In order to achieve the desired speed-up of the algorithm, the strategies outlined in this section were enacted. The data precision was reduced in order to reduce the size of the data and computation required. The hardware utilisation was increased by paralleling the CPU and FPGA. Data transfer times were addressed by improving data locality and using explicit actions. The compute time was reduced by merging multiple functionalities into a single FPGA kernel, allowing them to be executed in parallel, and replicating hardware. Lastly, optimal library parameters were found to balance the trade-off between image accuracy and performance.

4.1 Data Precision

One simple optimisation that can be made is to reduce the precision of the data. Originally in double precision (64 bits per floating point value), the data could be reduced to a single precision representation (32 bits per floating point value). From image analysis it was apparent that the extra bits did not affect the final images noticeably, therefore the extra effort required to compute with more bits was not necessary. Quicker mathematical operations on fewer bits and shorter data transfer times lead to shorter execution times and quicker build times for FPGA synthesis, making it a more convenient choice too. Lastly the resource demands are exponential in the number of bits used in the representation.

When making this change, it is important that it does not result in a loss for the integrity of the image. Using the metrics outlined in Section 2.5, the images from the reference and a single precision implementation of XD-GRASP were compared. Doing so revealed there is minimal difference in the accuracy of the reconstruction, with a strong SSIM score of 0.9971, MSE of 0.7367 and nMSE of 0.00001124.

4.2 Maximising CPU Utilisation

As revealed by the performance model (Section 3), the computation of the non-uniform Fast Fourier Transform was accountable for approximately 64.96% of the CPU execution time (the calculation for which is covered in Section 3). The goal became to minimise the proportion of the overall execution time that was accounted for by non-NUFFT computations. To achieve this, other parts of the algorithm were accelerated using an FPGA and were executed in parallel with the NUFFT. Therefore, the total time spent on the NUFFT acts as a lower bound to the total execution time. This means, if the NUFFT is accelerated, the XD-GRASP system proposed by this project will accommodate these accelerations, bringing down the total execution time with no further changes necessary. The bounds for this are discussed in Section 6.

There are, inevitably, times when the CPU cannot be computing NUFFT results as the input for the operation is not yet available. To minimise this stalling time, the workload of appropriate FPGA kernels were reduced such that they output the minimum data required to allow a NUFFT operation to begin (in cases where the kernel output was one of the NUFFT inputs). Not only does this reduce the idle time of the CPU, but it also increases the proportion of the time the FPGA and CPU are computing simultaneously, meaning the time cost of the FPGA can be covered by the NUFFT time, bring down the length of the overall execution.

Once CPU computation time was maximised, and consequently a majority of FPGA work occurring at the same time as the NUFFT, the proportion of algorithm time spent performing the NUFFT rose to 99.77% (as anticipated by the performance model).

4.3 Explicit Actions

When invoking kernels from the CPU, a high-level MaxCompiler API, called *actions*, provides an interface to control the FPGA. With it are checks for the validity of actions, including the number of ticks for the kernel, the input data and the output data. Actions result in the overheads of communications (data transfer) and computation (kernel functionality) being incurred simultaneously in one call and carry the overhead of checking the necessary parameters are set for each kernel call.

On the other hand, explicit actions were used in order to reduce the total FPGA time. By using these, the data transfer of one kernel task can be executed in parallel with the computation of another kernel task, thus reducing the overall time required for the work to be done on the FPGA. Further, there are no validity checks, reducing the overheads associated with a kernel invocation.

4.4 Flatten, Merge, Repeat

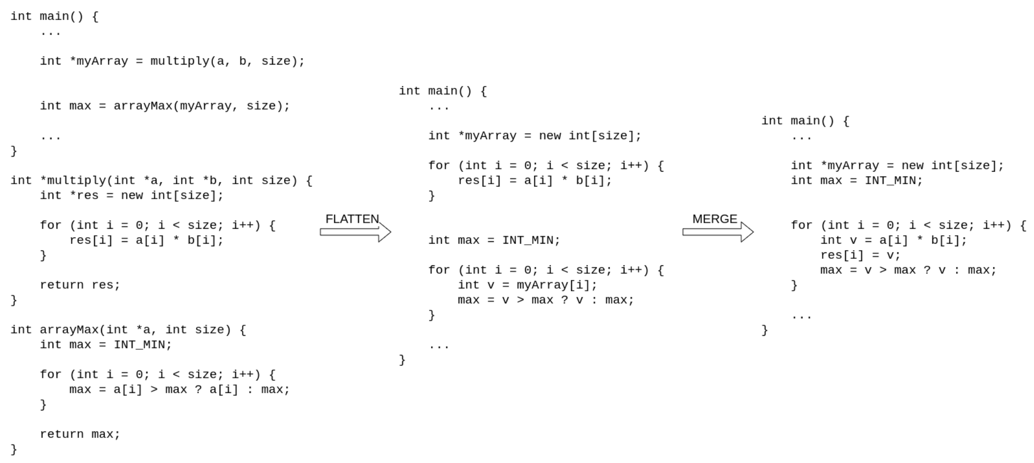
Following on from the CPU utilisation optimisations, when determining the functionality required for the FPGA kernels, it became important for the FPGA to compute as much as possible whilst the NUFFT was executing. To achieve this, the following steps were applied to the original C++ code:

1. Flatten: Inline methods, starting with the *deepest* method calls for chains of nested calls
2. Merge: Combine adjacent operations. This includes, but is not limited to:
   (a) Equivalently sized iterations over data
   (b) Evaluating the effect of multiple transformations on the data to produce one transformation which has the same overall effect
3. Repeat: Flatten and merge again, until no further code reductions are possible

This kind of optimisation is not possible for the original MATLAB implementation, unless all of the operators are re-implemented, showing the necessity of a C++ implementation of XD-GRASP in addition to the computation overhead of MATLAB itself.

The optimisations compound over iterations, as the flatter code can become simpler (in terms of the number of statements), making it easier to merge with other code. In some cases for XD-GRASP, such as for the total variation operator (the understanding of which is not required for this project), the effective transformation could be computed by hand to reduce a two step process to one.

All of this, however, comes at a cost. The resulting code is often unrecognisable from the original algorithm as the boarders between mathematical operations become blurred. Whilst this puts maintainability at risk, this optimisation opens the door to a similar process for FPGA kernels.



```
int main() {
    ...

    int *myArray = multiply(a, b, size);


    int max = arrayMax(myArray, size);

    ...
}

int *multiply(int *a, int *b, int size) {
    int *res = new int[size];

    for (int i = 0; i < size; i++) {
        res[i] = a[i] * b[i];
    }

    return res;
}

int arrayMax(int *a, int size) {
    int max = INT_MIN;

    for (int i = 0; i < size; i++) {
        max = a[i] > max ? a[i] : max;
    }

    return max;
}
```

```
int main() {
    ...

    int *myArray = new int[size];

    for (int i = 0; i < size; i++) {
        res[i] = a[i] * b[i];
    }


    int max = INT_MIN;

    for (int i = 0; i < size; i++) {
        int v = myArray[i];
        max = v > max ? v : max;
    }

    ...
}
```

```
int main() {
    ...

    int *myArray = new int[size];
    int max = INT_MIN;


    for (int i = 0; i < size; i++) {
        int v = a[i] * b[i];
        res[i] = v;
        max = v > max ? v : max;
    }

    ...
}
```

Figure 12: A simple case of flattening and merging for example code.

FPGAs are able to perform computations, even within the same kernel, simultaneously. To fully utilise this dataflow parallelism, the kernels' functionalities were increased to include extra computation for purposes beyond the original use of the kernel. For example, when computing an array result, the kernel may accumulate the sum of the elements in the resulting array in parallel, to avoid a second iteration over the data. The 'extra' functionality pushed onto a kernel arose from a similar flattening approach as was used for the C++ code. However in this case, rather than merging loops of code, kernels with similar iteration patterns over the same data had their functionalities merged, outputting two results in parallel.

4.5 Data Locality

As learned from the performance model, the communication cost between the CPU and FPGA incurs a significant time penalty, often with it dominating the time for an FPGA task to complete. In order to overcome this, the frequently-used, fixed datasets required by XD-GRASP had to be persisted as-close-as-possible to the FPGA compute hardware. For the FPGA architecture, this meant persisting data in large memory (DRAM). Not only does this data persistence reduce the amount of data transported per kernel invocation, but it also frees PCIe bandwidth for the, reduced, data to use.

In most of the kernels which use LMem-persisted datasets, there is always an equally-sized dataset streamed from the CPU. This means that the time taken to transfer data from LMem to the kernel is covered by the time taken to transfer data from the CPU to the kernel as the architecture has been optimised such that both transfers occur at the same time. Since the datasets are the same size, this property can be confirmed by a comparison of the bandwidths from CPU to FPGA (via the PCIe bus) and from DRAM to B/URAM. These bandwidths are 15750MB/s and 45000MB/s. Therefore, it was possible to overlook double buffering here (Section 2.6) as it did not yield any benefit. The exception to this is `MultiplicationKernel` which has both inputs streamed from LMem. However this is called relatively few times and has a low time cost. Thus double buffering its inputs and outputs was not a priority.

In addition, the data stored in LMem is organised such that the reads are sequential, the most efficient access pattern, avoiding the cost of random access.

Whilst it would have been most optimal to store the datasets in FMem, due to the quicker access time than LMem, the memory requirements of the kernel computations were prioritised over persisting data, as this would allow for greater replicated hardware (Section 4.6). Further, there is an FMem requirement for the FIFO queues used for dataflow scheduling. In order for a kernel to compute, it requires its inputs at the same time. In order to ensure the inputs are ready simultaneously, FIFO queues are used to equalise the latencies of the inputs, as demonstrated in Figure 13.

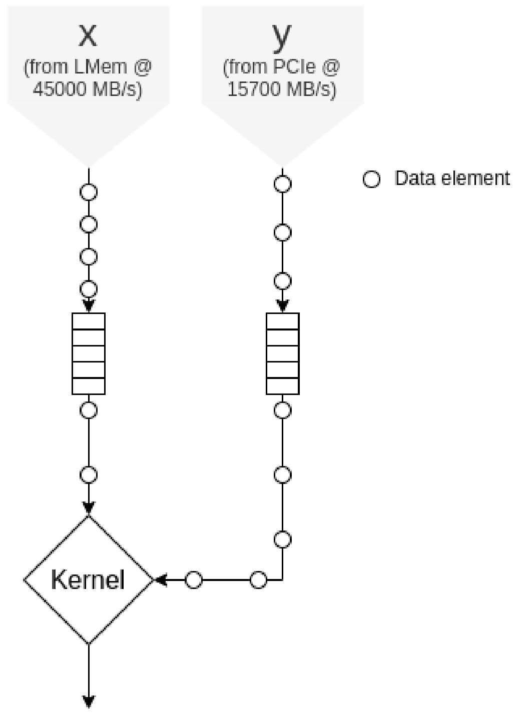


Figure 13: The use of FIFO queues by an FPGA to ensure data arrives at a kernel at the same time.

4.6 Replicating Hardware

Whilst porting sections of code onto an FPGA brings one level of speed-up, this factor can become multiplicative if the hardware of the kernel is replicated. In the architecture design, a parameter, `pFactor`, is defined for each kernel, dictating the level of hardware replication in the kernel, thus increasing the parallelism. For `pFactor = p`, the compute hardware of the FPGA kernel is replicated `p` times and `p` elements are read from the input stream per tick. The latter operation requires that the `pFactor` must then also be a factor of the number of elements in the input stream. For XD-GRASP, this meant padding arrays to fit this factor.

The effect of various `pFactor`s were predicted using the performance model (Section 3.5) and thus an optimal allocation of DSPs and FMem could be allocated to kernels which completed the slowest. Thus the approximate compute time previously stated (Equation 12) becomes:

$$T_c \approx \frac{E}{\texttt{pFactor} \cdot C \cdot E_c} + \textit{Fill Time} \tag{15}$$

Where E is the number of elements to be processed, E_c is the number of elements processed per clock cycle and C is the clock frequency, in Hertz. In some cases, increasing the `pFactor` increased the memory (B/URAM) requirement of the kernel as memories had to be replicated to allow for parallel reads and writes. Future work may entail mitigating this effect.

4.7 Optimal Parameters for GPU

In an effort to push the acceleration, a GPU was added to the system to increase the NUFFT performance. The library used, gpuNUFFT [13], offers a variety of parameters to fine tune the library to the use case. Experiments with the parameters, the results of which are shown in Table 2, highlighted a performance/accuracy trade-off which could be taken advantage of. As the result show, a kernel width of 1 (the minimum it can be) lead to the greatest performance increase, but consequently at the largest accuracy cost. What is, perhaps, surprising is the accuracy offered by a kernel width of 3 and an oversampling factor of 2, whilst bringing comparable performance. Whilst the most optimal, with regard to time, configuration could have been chosen, the final configuration (highlighted with in bold in Table 2) offers a compromise of high accuracy and performance.

Note that in measuring the 'NUFFT Time' during XD-GRASP under the configuration, the cumulative anticipated time spent performing both types of the NUFFT on representatively-sized data was calculated from benchmarked NUFFT types on the same hardware used in the performance evaluation (Section 6.3).

KW	OSF	SW	MSE	nMSE	SSIM	Total Time
1	1.25	16	3.3308	0.0001	0.9322	8.5147
1	1.25	24	3.3407	0.0001	0.9319	8.8257
1	2.0	16	3.3967	0.0001	0.9173	8.8512
1	1.25	8	3.3308	0.0001	0.9322	8.9768
1	2.0	24	3.3967	0.0001	0.9173	9.3052
1	2.0	8	3.3967	0.0001	0.9173	9.3349
3	**2.0**	**16**	**0.9791**	**0.0000**	**0.9918**	**9.7737**
3	1.25	16	1.8403	0.0000	0.9708	9.8679
3	1.25	8	1.8403	0.0000	0.9708	10.3848
3	2.0	8	0.9791	0.0000	0.9918	10.4832
3	1.25	24	1.84	0.0000	0.9708	10.5530
3	2.0	24	0.9791	0.0000	0.9918	11.2008
7	2.0	16	1.2521	0.0000	0.9915	16.4491
7	2.0	8	1.2521	0.0000	0.9915	16.6098
7	1.25	8	3.2759	0.0000	0.9627	18.3329
7	2.0	24	1.2521	0.0000	0.9915	18.3366
7	1.25	16	3.2758	0.0000	0.9627	18.5891
7	1.25	24	3.2759	0.0000	0.9627	20.7342

Table 2: The NUFFT benchmarks, along with SSIM scores, for different gpuNUFFT configurations of kernel widths (KW), sector widths (SW) and oversampling factors (OSF), ordered by 'Total Time'.

It can be seen that the biggest influence of time is the kernel width, with it consistently being inversely proportional to the time. There does not seem to appear to be a strong connection between the other parameters and the accuracy or time and thus once a kernel width of 3 was selected as a compromise between accuracy and speed, the most accurate configuration was chosen. The respective scores for the pixel intensity differences, MSE and nMSE, were all small enough to disregard and use SSIM (Section 2.5) as the sole indicator of accuracy. The parameter search space was as follows:

- Kernel width: 1, 3, 7
- Oversampling factor: 1.25, 2.00
- Sector width: 8, 16, 24

It is worth noting that, during the parameter search, the MSE and SSIM scores reflect a closeness to the reference, reconstructed using its own NUFFT parameters. Weaker scores do not necessarily reflect a poor image, as can be confirmed by qualitative analysis, however stronger scores provide more robust evidence that the image quality has not been compromised.

By and large, the optimisations discussed in this section are applicable to other algorithms being accelerated on an FPGA, with the exception of the optimal GPU parameters, which is more to specific to projects making use of external libraries.

5 Implementation

With the optimisations to XD-GRASP outlined in Section 4, this section describes how the various hardware was used to build the accelerated XD-GRASP system, capable of reconstructing 320x320 images for 8 respiratory phases.

Implementing the two types of NUFFT on an FPGA would have been infeasible for this project's time-frame and therefore existing alternative accelerators for it, namely a GPU, were sought out instead. Further, the NUFFT entails a convolution which, if implemented via a matrix multiplication, GPUs become unbeatable at executing, due to their SIMT architectures.

5.1 Systems

Using a CPU for the NUFFT operations and an FPGA for everything else, the algorithm was spread across these components and coordinated by the CPU to make the two-prong heterogeneous system. As mentioned in Section 4, one large factor in achieving the speedup was to utilise the CPU as much as possible.

The CPU triggers a set of asynchronous FPGA tasks, which are executed sequentially on the FPGA, and waits for the completion of tasks before performing NUFFT operations. Following the completion of a single NUFFT, the CPU then triggers the next FPGA tasks (corresponding to the output of the NUFFT) and, again, waits for the result. In some cases, the CPU waits for a full batch of FPGA tasks to complete, as there is no available work to do (NUFFTs).

The addition of a GPU does not change the orchestration mechanics of the two-prong heterogeneous system. In this case, rather than performing the NUFFT itself, the CPU unloads this work onto the GPU synchronously, to give a three-prong heterogeneous system. There is no benefit to asynchronous NUFFT calls as the execution time is still dominated by the NUFFT. Thus a simple swap of NUFFT operator is required, with no further changes to the code.

To produce a 4D reconstruction (with dimensions in width, height, depth and time), 2D images (width and height) are reconstructed for all the respiratory phases (the time dimension) at once. The missing dimension (depth) is obtained through sequential runs of the 2D+time system on data for different slices of the scan data.

The pre-processing step of XD-GRASP has been omitted as effort is better spent accelerating the part of the algorithm which is executed for every slice (2D image) of the 4D reconstruction.

An aerial view of the FPGA implementation, produced by MaxCompiler, is available in Appendix A.

5.2 Coordination

In order to orchestrate the concurrent computation of the FPGA and CPU, a synchronisation primitive was added (Listing 1) which allowed FPGA tasks to be called asynchronously and the resources of which to be freed once waited-for. Using this, the CPU could batch-trigger FPGA tasks and then proceed to begin the NUFFT operations as soon as possible, using `async_task_t` to ensure the input to the NUFFT was ready.

```
typedef struct {
    // The thread to wait for
    std::thread *run;

    // The arrays to be freed once `run` completes
    void **freeableDependents;

    // The size of the freeableDependents entry
    int nFreeableDependents;

    // Ad-hoc identifier to instruct how to free an array
    int *dependentsTypes;
} async_task_t;
```

Listing 1: Synchronisation primitive

5.3 Kernel Breakdown

This subsection gives an overview of the kernel functionalities and the parametric equations used by the performance model (Section 3) to characterise their performance.

There are four inputs to XD-GRASP (excluding data required for any pre-processing steps) to consider:

- `b1`: Coil sensitivities for the MRI scanner the signal is measured by
- `kdatau`: The sorted (by respiratory phase) raw signal data

- `ku`: The sorted (by respiratory phase) k-space trajectory
- `wu`: The sorted (by respiratory phase) density compensation weights

Note that the `u` suffix reflects data which has been sorted by respiratory phase and when the above datasets are used by kernels, access to them is offset to allow the appropriate values to be read.

In addition, there are several dimensions are defined here, to be used in the mathematical definitions of kernels:

- $ntres$: The number of respiratory phases for the reconstruction
- nc: The number of receiver coils for MRI data acquisition
- $nSamples$: The number samples acquired by each coil for one respiratory phase ($nSamples = nx \cdot nline$)
- $nPixels$: The number of pixels in the image ($nPixels = npx \cdot npy$)
- nx: The number of samples per spoke acquired in radial sampling
- $nline$: The number of spokes acquired in radial sampling
- npx: The horizontal pixel dimension of the image
- npy: The vertical pixel dimension of the image

As the kernels combine maximal functionality between NUFFT operations, there is often not a direct correspondence between the computation of a kernel and a step in the XD-GRASP algorithm.

Two of the more complex kernels are outlined below. The rest of the kernel details are in Appendix B.

5.3.1 `CombineAcrossCoilsKernel`

During the transformation from k-space to image space, the measurements taken across multiple receiver coils during one respiratory phase on the MRI scanner are combined by this kernel. The input data, `x` is transposed before the computation is performed, making use of FMem to do so. To achieve this, there are nc separate memories in use, to allow nc parallel reads and writes.

$$\texttt{out}_i = \frac{\sum_{c=0}^{nc} |\texttt{b1}_{i,c}|^2}{\sum_{c=0}^{nc} \texttt{tmp}_{i,c} \cdot \texttt{b1}_{i,c}} \cdot C$$

$$\texttt{tmp} = \texttt{x}^\top$$

$$C = \begin{cases} \frac{nx \cdot \pi}{2 \cdot nline} & \texttt{halve === 1} \\ \frac{nx \cdot \pi}{nline} & \text{otherwise} \end{cases}$$

- **Input:**
  - `x`: The output of a type 1 NUFFT across all coils, for a respiratory phase
  - `b1`: As stated above
  - `halve`: A flag to dictate whether or not each coefficient of the result is halved (halving is enabled if `halve === 1`)
- **Output:**
  - `out`: Image space data for a respiratory phase
- **Hardware requirements:**
  - DSPs: 167 (does not support variable `pFactor`)
  - FMem: 6.5536MB

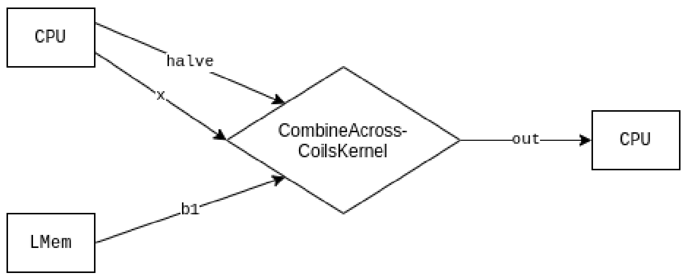


Figure 14: The dataflow of `CombineAcrossCoilsKernel`.

5.3.2 `TransposedMultiplicationKernel`

As the name suggests, the inputs and multiplied together before being output in a transposed ordering. In addition, the temporal variance between `x` and `xNext` is output.

$$
\begin{aligned}
\texttt{out}_{i,c} &= \texttt{tmp}_{i,c} \cdot \texttt{b1}_{i,c} \\
\texttt{tmp} &= \texttt{x}^{\top} \\
\texttt{TVOut} &= \sum_{i=0}^{nPixels} \sqrt{|v \cdot \bar{v}| + \texttt{l1Smooth}} \\
v &= \begin{cases} 0 & \texttt{r === } ntres \\ \texttt{xNext} - \texttt{x} & \text{otherwise} \end{cases}
\end{aligned}
$$

- **Input:**
  - `x`: Image space reconstruction for respiratory phase r
  - `xNext`: The reconstruction for respiratory phase $r_{Next} = \min(r + 1, MAX\_RESP\_PHASE)$
  - `b1`: As stated above
  - `r`: The index of the current respiratory phase ($= r$)
  - `c`: The index of the current coil
  - `l1Smooth`: A conjugate gradient smoothing parameter
- **Output:**
  - `out`: The element-wise multiplication of `x` and the corresponding coil entry of `b1`
  - `TVOut`: The temporal variance between `x` and `xNext`
- **Hardware requirements:**
  - DSPs: $19 \times \texttt{pFactor} + 1$
  - FMem: $\approx$6.5536MB*

*As pFactor grows, the number of reads per kernel memory increases causing MaxCompiler to replicate the memory during compilation. This is a side effect to be mitigated in the future however, as such, means a definite relationship for the FMem requirement and pFactor cannot be given.

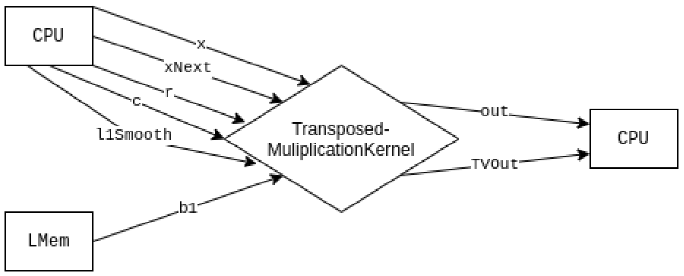


Figure 15: The dataflow of `TransposedMultiplicationKernel`.

The kernels can be summarised as follows:

Kernel	DSP Eq.	Memory Req. (MB)
`CombineAcrossCoils`	167	6.5536
`Grad`	$42 \times$ pFactor	0
`Multiplication`	$6 \times$ pFactor	0
`Objective`	$12 \times$ pFactor	0
`PostNUFFTType2`	$12 \times$ pFactor	0
`TransposedMultiplication`	$19 \times$ pFactor $+ 1$	$\approx$6.5536
`Update`	$24 \times$ pFactor	0

Table 3: A summary of the implemented kernels.

5.4 Dataflow

The dataflow for the algorithm implementation is shown here for the three-prong system. In order to understand the dataflow for the two-prong system, with no GPU, the division between CPU and GPU can be removed and these sections merged.

Note that in these diagrams, crossing a dashed line denoting the boundary between CPU and FPGA means a PCIe data transfer. Only more significant datasets are shown.

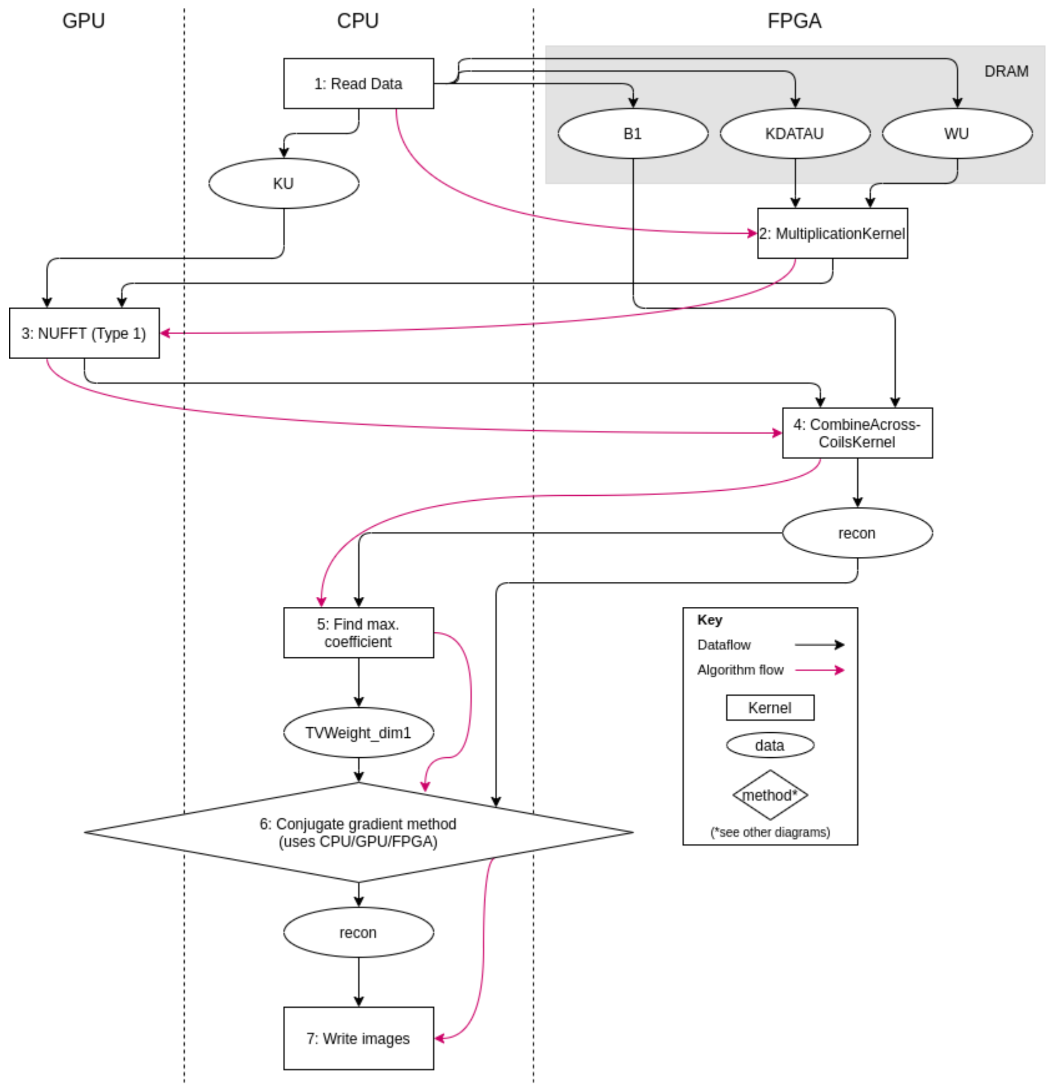


Figure 16: The overall system dataflow.

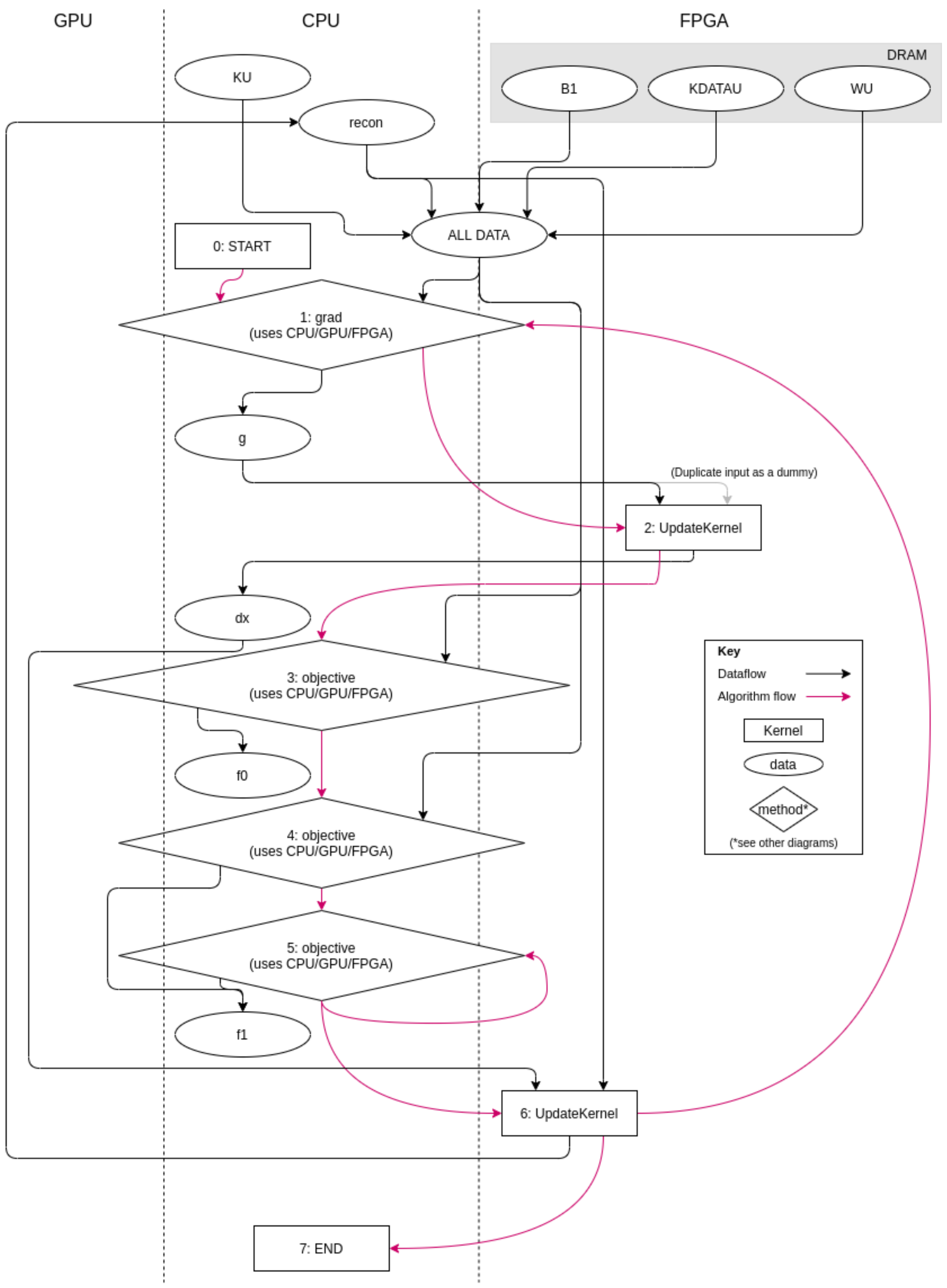


Figure 17: The conjugate gradient method dataflow.

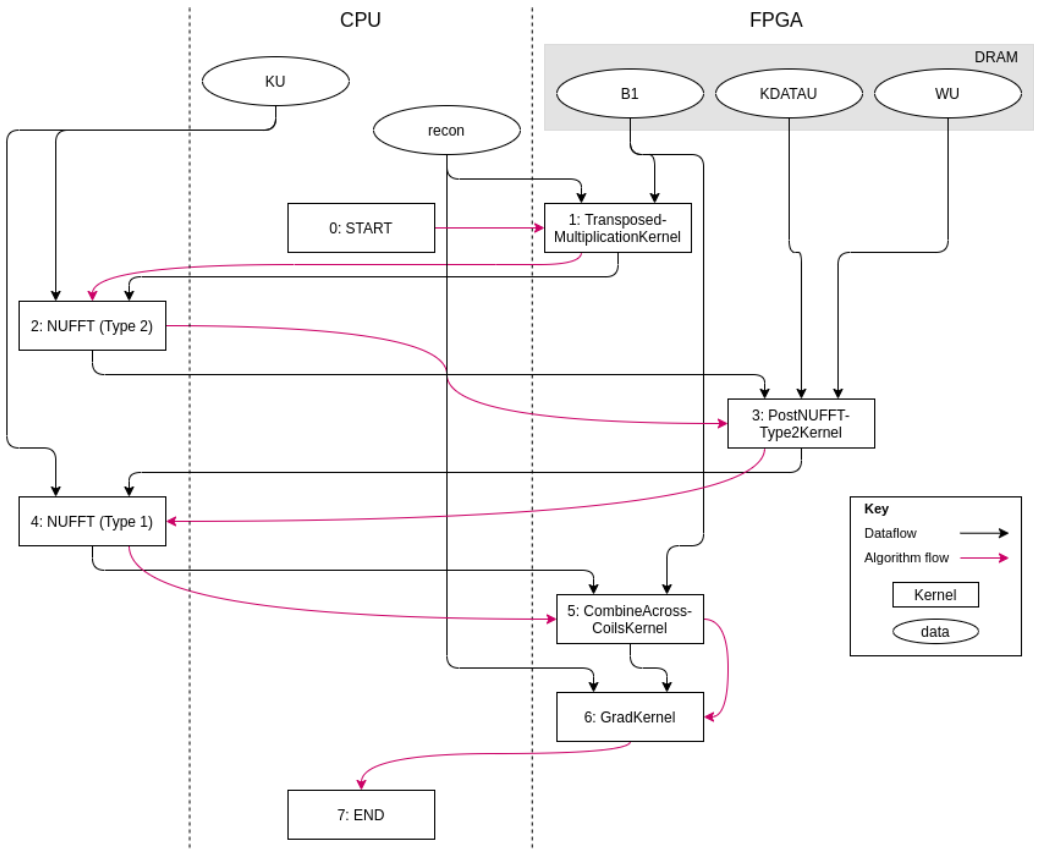


Figure 18: The gradient method dataflow.

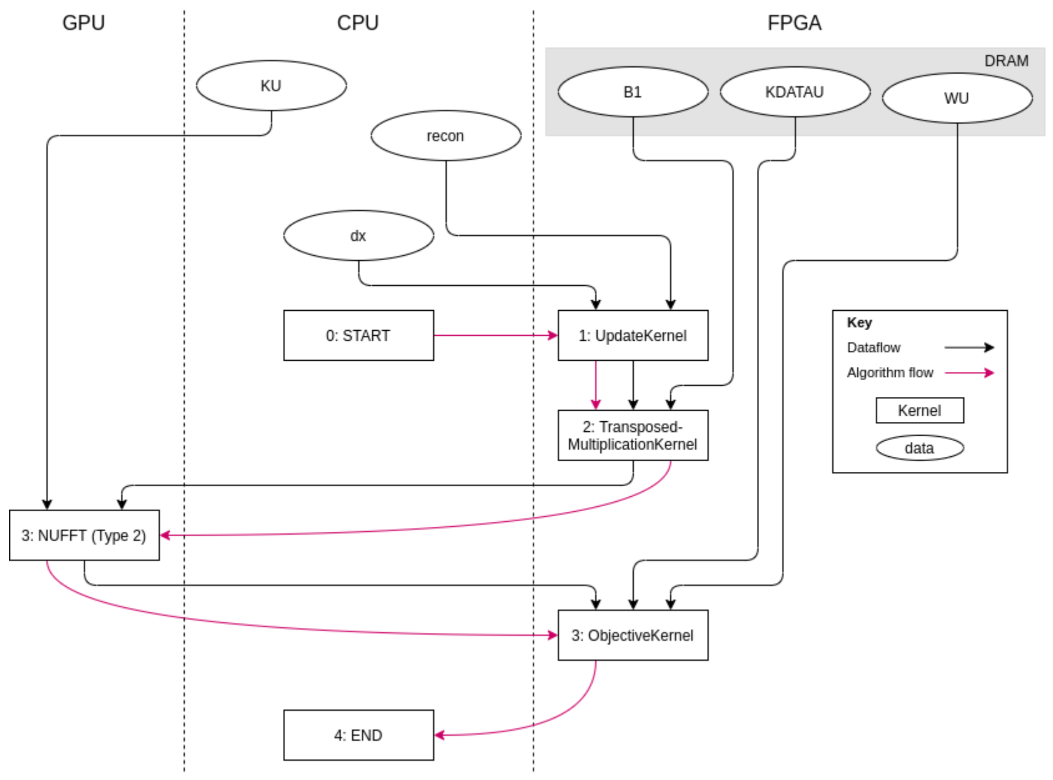


Figure 19: The objective method dataflow.

This system, both the two- and three-prong implementations, is reflected in the models (Section 3). These implementations will accommodate a faster NUFFT, should one be added to the project, as the `async_task_t` mechanism prevents operations executing without the necessary input. The kernels often compute two functions, in parallel, as a result of the 'Flatten, Merge, Repeat' optimisation outlined in Section 4.

6 Performance Evaluation

In order to assess the success of this project, both the accuracy of the reconstruction and the computation time must be considered. A solution which produces clear images in a satisfiably short amount of time is regarded as successful.

Results in this section consider the reconstruction of a 2D image, for 8 respiratory phases. For each implementation below, the expected 4D reconstruction time can be calculated by multiplying the given reconstruction time by the depth of the 3D image.

There are four classes of algorithm implementation to consider (to be referred to by letter ID):

(a) Demo MATLAB implementation (the reference)

(b) Homogeneous C++ implementation

(c) CPU/GPU implementation

(d) Two-pronged CPU/FPGA implementation (see Section 5.1)

(e) Three-pronged CPU/FPGA/GPU implementation (see Section 5.1)

Performance is evaluated using two datasets:

- Dataset 1: The demo data included in the XD-GRASP demo code (Feng et al., [35]) (producing images with dimensions 256x256)
- Dataset 2: Data generously provided by the Institute of Cancer Research [36] (producing images with dimensions 320x320)

6.1 Benchmarking

First, the methods for measuring the execution of the FPGA, and consequently using these results to compute an execution time for XD-GRASP, need to be outlined.

Each kernel was executed and timed independently 200 times during each benchmark run, and, after 8 benchmark runs, the mean execution time and variance could be calculated.

The benchmarks for the CPU and GPU NUFFT operations were measured similarly.

Using these mean times and the number of invocations of each kernel for XD-GRASP and the NUFFT, whether on the CPU or GPU, the total execution time of XD-GRASP was calculated. This calculation followed the performance model calculations closely, with a simple replacement of kernel execution times producing a valid result for implementations (d) and (e).

The NUFFT benchmarks are displayed here (Table 4). The kernel benchmarks are discussed in Section 6.4.

	CPU (FINUFFT)	GPU (gpuNUFFT)
Type 1 Time (s)	0.00258800	0.00186400
Type 2 Time (s)	0.00247200	0.00146900

Table 4: Benchmarks for the NUFFT on the CPU and GPU

To see whether or not the GPU NUFFT could be accelerated further, benchmarks for a more powerful GPU, a NVidia® Titan Xp, were obtained. However, the gpuNUFFT library is not optimised, meaning changing the GPU leads to no performance improvement. The GPU utilisation for both the Titan Xp and 1080 Ti is around %39 for the type 1 NUFFT and %19 for the type 2 NUFFT. Thus a 1080 Ti was used as a machine was available with it with no other users, meaning less interference during testing.

In theory the multiplicative cost of a type 1 NUFFT is 2.66E+06 and 1.11E+06, approximated using the C source for exponential operations[2]. Taking this, and the 1080 Ti's 345.4 GigaFLOP capabilities, give a theoretical NUFFT computation time of 7.51E-7 (type 1) and 3.14E-7 (type 2) seconds. In reality, the communication cost, which is omitted from the previous calculation, increases this time. This theoretical time demonstrates the potential of the GPU used, with an over estimation in the communication cost still leading to NUFFT times three orders of magnitude smaller than those measured.

6.2 Image Accuracy

As the output of XD-GRASP is intended for use in sensitive decisions, such as radiation dose planning, the clarity and validity of the produced images should not be compromised. The measures outlined in Section 2.5 become of

[2]https://code.woboq.org/userspace/glibc/sysdeps/ieee754/ldbl-128/e_expl.c.html

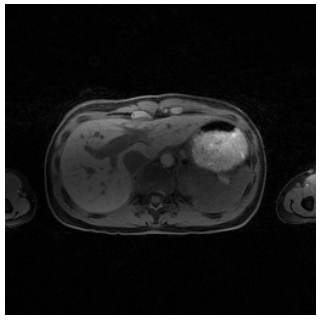
use here, providing statistical evidence that the algorithm has been correctly implemented and optimisations have not led to incorrect reconstructions.

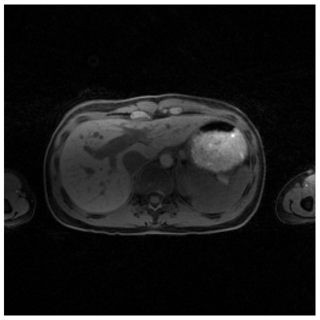
Whilst the inconsistencies between different hardware's floating point handling lead to small differences, it can be seen from the results (Table 5) that this bears minimal influence on the output. Values are produced from the mean of the scores across all respiratory phases. Results are not available for configurations utilising an FPGA with Dataset 1 as the implementation and optimisations are specific to the dimensions of Dataset 2. To adjust the XD-GRASP implementation for the dimensions of Dataset 1, the FPGA implementation would have to be redesigned to accommodate the different data sizes, since the hardware utilisation has been maximised for a specific set of dimensions.

Impl. ID	Dataset 1 (Demo)			Dataset 2		
	MSE	nMSE	SSIM	MSE	nMSE	SSIM
(a) (ref.)	0	0	1	0	0	1
(b)	0.4500	0.00000687	0.9999	0.7365	0.00001124	0.9971
(c)	0.9791	0.00001494	0.9906	1.2601	0.00001923	0.9879
(d)	N/A			0.7369	0.00001124	0.9971
(e)[†]				1.2601	0.00001923	0.9879

Table 5: Metrics (Section 2.5) collected for various implementations of XD-GRASP. ([†]Practical results not obtainable, see Section 6.6.)

Qualitative assessment of the images gives similar results as can be seen in Figures 20, 21.


(a) SSIM = 1


(b) SSIM = 0.9999

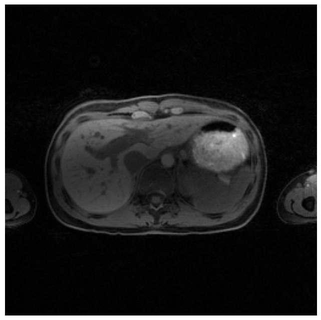
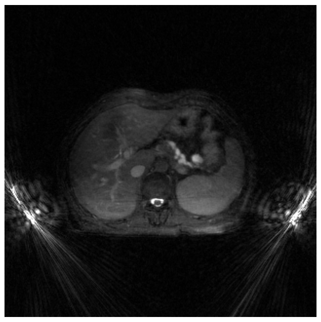
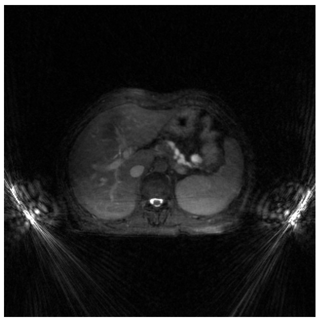
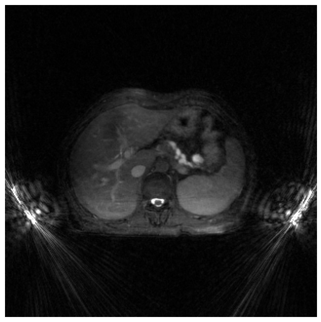

(c) SSIM = 0.9906

Figure 20: The comparisons of the XD-GRASP implementations, outlined in Section 6, with the corresponding SSIM scores when compared to the reference, (a), for Dataset 1. Results are not available for configurations utilising an FPGA with Dataset 1 as the implementation is specific to the dimensions of Dataset 2.


(a) SSIM = 1


(b) SSIM = 0.9971


(c) SSIM = 0.9879

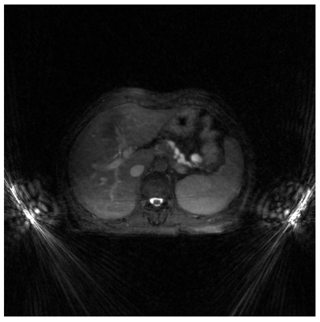

(d) SSIM = 0.9971

Figure 21: The comparisons of the XD-GRASP implementations, outlined in Section 6, with the corresponding SSIM scores when compared to the reference, (a), for Dataset 2.

As can be seen from the image comparisons, the accuracy of the reconstructions has not been compromised during acceleration, with qualitative judgement matching quantitative analysis.

6.3 Performance Improvements

In order to achieve real-time adaptive radiotherapy, it is necessary that the algorithm running on an FPGA completes in a satisfiably short amount of time and meets real-time requirements. Table 6 gives an overview of the execution times. For the CPU benchmarks, a 12-core 14nm Intel® Core i7-8700K was used, running at 3.20GHz with boost clock frequency 4.3GHz. For GPU results, a NVidia® GeForce GTX 1080 Ti was used. Lastly, the FPGA used in the performance model was the Xilinx® VU9P running at 200MHz.

In order to produce a valid comparison, any early-stopping in backtracking line search and the conjugate gradient method was disabled, forcing the worst-case execution time. This ensures that, across all the implementations, equivalent work is done. In practical use, early-stopping is re-enabled, leading to shorter execution times.

It was not possible to assemble a system with the benchmark CPU and FPGA, due to the Covid-19 pandemic and lockdown measures, so the performance model, along with kernel execution time benchmarks were used to calculate the execution time for (d) and (e) (details in Section 6.1).

Impl. ID	Description	Reconstruction Time (s)
(a)	MATLAB (ref)	65.118
(b)	CPU	24.722
(c)†	CPU/GPU	18.435
(d)*	CPU and FPGA	16.071
(e)†	CPU, GPU and FPGA	9.7836

Table 6: The reconstruction time of different XD-GRASP implementations. (*See Section 6.1.calc) (†Practical results not obtainable, see Section 6.6.)

The speed increase from MATLAB to pure C++ is over 2.6x, consistent with the expectation from the finer-grained control enabled by the language. Further still, utilising an FPGA, with the NUFFT on the CPU, resulted in a 1.54x speed-up against the CPU implementation, with the NUFFT being the bottleneck. A more detailed breakdown of the execution times shown in Table 7 reveals that 0.06% of the total execution time is incurred solely by the FPGA, with the bulk of the time coming from the NUFFT execution on CPU – one of the aims during optimisation.

In an effort to reduce the NUFFT time requirement further, the addition of a GPU, using optimal parameters discussed in Section 4, pushes the speed-up to 2.53x (when compared with the CPU-only implementation) and 1.88x (when compared with the CPU/GPU implementation). Despite the performance increase with the extra hardware, the execution of the algorithm is still dominated by the NUFFT time requirement, incurring the same non-NUFFT time cost as with CPU-based NUFFT.

Impl. ID	Time for One NUFFT		Total NUFFT Time	Total Non-NUFFT Time
	Type 1	Type 2		
(b)	0.002558	0.002472	16.06067	13.07212
(c)†	0.001864	0.001469	9.77370	13.07212
(d)*	0.003307	0.002035	16.06067	0.00991
(e)†	0.001864	0.001469	9.77370	0.00991

Table 7: The reconstruction time breakdown different XD-GRASP implementations. (*See Section 6.1) (†Practical results not obtainable, see Section 6.6.)

The performance of the proposed system, implementation (e), when reconstructing 2D+time images can be extrapolated to find the worst-case reconstruction time for 4D reconstruction of an image depth of 147. In this case, the worst-case reconstruction time is 1436.73 seconds, around 24 minutes. Practically, this time is likely to be shorter as the early-stopping conditions of the algorithm are re-enabled. As mentioned, this performance can be pushed further through improvement of the GPU NUFFT implementation.

6.3.1 Armdahl's Law

Armdahl's Law can be used to measure the speed-up of the acceleration:

$$\begin{aligned}\text{Original time of accel'd code} &= (\text{Total exec. time}) \\ &\quad - (\text{Total NUFFT time}) \\ &= 24.72233 - 16.06067 \\ &= 8.66166\end{aligned} \tag{16}$$

$$\begin{aligned}\text{Speed-up of accel'd code} &= \frac{\text{Original time of accel'd code}}{\text{Accel'd time of accel'd code}} \\ &= \frac{8.66166}{0.11223} \\ &= 77.18\end{aligned} \tag{17}$$

Using 17, the theoretical speed-up and limit of theoretical speed-up can be obtained:

(For CPU NUFFT)

$$\begin{aligned}\text{Prop'n of time occupied by accel'd code} &= \frac{\text{Non-NUFFT Time}}{\text{Total Time}} \\ &= \frac{8.66166}{24.72233} \\ &= 0.3504\end{aligned} \tag{18}$$

$$\begin{aligned}\text{Theoretical speed-up} &= \frac{1}{(1-0.3504)+\frac{0.3504}{77.18}} \\ &= 1.5286\end{aligned} \tag{19}$$

$$\begin{aligned}\text{Limit of theoretical speed-up} &= \frac{1}{1-0.3504} \\ &= 1.5393\end{aligned} \tag{20}$$

(For GPU NUFFT)

$$\begin{aligned}\text{Prop'n of time occupied by accel'd code} &= \frac{\text{Non-NUFFT Time}}{\text{Total Time}} \\ &= \frac{8.66166}{18.43535} \\ &= 0.4698\end{aligned} \tag{21}$$

$$\begin{aligned}\text{Theoretical speed-up} &= \frac{1}{(1-0.4698)+\frac{0.4698}{77.18}} \\ &= 1.8648\end{aligned} \tag{22}$$

$$\begin{aligned}\text{Limit of theoretical speed-up} &= \frac{1}{1 - 0.4698} \\ &= 1.8862\end{aligned} \tag{23}$$

According to Armdahl's Law [37], applied to the CPU and FPGA implementation (d), in comparison with the C++ implementation (b), the theoretical speedup is 1.4698. Further, the limit of the theoretical speed-up is 1.5393, which the observed speed-up (1.5241) lies beneath. If the data transfer times are omitted, the anticipated speed-up rises to 1.5383. The calculations for this are shown in Equations 18, 19 and 20.

As a machine with the same CPU and GPU used for benchmarking was not available (discussed in Section 6.6.), Armdahl's Law can only be applied as an approximation. In this case, the theoretical speed-up is 1.7502 and the limit of the theoretical speed-up is 1.8862. The observed speed-up is 1.8558. If the data transfer times are omitted, the anticipated speed-up rises to 1.8843. The calculations for this are shown in Equations 21, 22 and 23.

The current system is limited by the performance of the NUFFT as this is not accelerated by an FPGA. This would be the next focus for optimisation when accelerating XD-GRASP further. Even with the maximum speed-up attainable, the system cannot achieve real-time reconstruction as the time demand of the NUFFT is too high.

If, however, the NUFFT operations completed in 0 time, the total execution time becomes 0.1122s, a speed-up of 580x over the MATLAB implementation and 220x over the C++ implementation, satisfying real-time reconstruction. This suggests that no additional FPGA acceleration is required and supports that the NUFFT should be accelerated next, to bridge the gap between the current execution time and desired real-time performance.

6.4 Kernel Performance

By benchmarking the executions of kernels, both the FPGA and CPU implementations, insight into the kernel performance can be obtained (see Table 8).

Kernel	CPU Bench. (s)	FPGA Bench. (s)	Perf. Model Expected (s)
`CombineAcrossCoils`	0.022510	0.00000778	0.00068675
`Grad`	0.007986	0.00000604	0.00000763
`Multiplication`	0.000038	0.00000710	0.00000891
`Objective`	0.000395	0.00000912	0.00001113
`PostNUFFTType2`	0.000076	0.00000995	0.00001113
`TransposedMultiplication`	0.002824	0.00000917	0.00010708
`Update`	0.000972	0.00000735	0.00002133

Table 8: A comparison of kernel benchmarks and expectations.

In most cases, the difference between the expected and observed kernel execution times is reasonably small and can be attributed to the generalisations the performance model makes about kernel execution. However, for `CombineAcrossCoils` the difference is surprisingly large and requires further investigation into why this kernel completes in such an impressive time, in comparison to the performance model expectation.

To ensure the validity of the FPGA kernel benchmarks, the variance for each kernel was computed (Table 9).

Kernel	FPGA Bench. (s)	Variance
`CombineAcrossCoils`	0.00000778	1.77E-12
`Grad`	0.00000604	7.39E-13
`Multiplication`	0.00000710	4.51E-12
`Objective`	0.00000912	4.58E-12
`PostNUFFTType2`	0.00000995	3.17E-12
`TransposedMultiplication`	0.00000917	1.70E-12
`Update`	0.00000735	2.17E-13

Table 9: The mean and variance for kernel benchmarks.

6.5 FPGA Utilisation

The hardware utilisation of the FPGA used in this project, the Xilinx® VU9P, is given in Table 10. Despite aiming for 80% resource utilisation, the actual utilisation of the hardware is relatively low. This is due to higher `pFactor` configurations causing the FPGA compilation to fail to meet timing at 200MHz without architecture specific optimisation aimed to shorten the critical path, the path hardware with the largest delay[3]. Meeting timing requires an FPGA layout to be found such that the hardware components, and the connections between them, can be placed in such a configuration that meets timing requirements, such as inputs of a hardware component arriving at the same time.

Component	Total Available	Total Used
LUTs	1182240	451266
FFs	2364480	803075
BRAMs	2160	1598
DSPs	6840	1865

Table 10: The FPGA hardware utilisation.

6.6 GPU Results

It was not possible to execute the three-prong heterogeneous system on hardware as there was not a machine with both GPU and FPGA capabilities available and, due to the Covid-19 pandemic and lockdown measures, it was not possible to access the data center to assemble one. Thus results for the image accuracy and performance were derived from measurements taken from a CPU/FPGA implementation (i.e. implementation (d)) and GPU benchmarks.

In the case of image accuracy, there appears to be negligible differences between the CPU and CPU/FPGA implementations ((b) and (d) respectively). Therefore, the image accuracy for a CPU/FPGA/GPU, (e), is approximated using the image accuracy of the CPU/GPU implementation.

On the other hand, to obtain a execution time for (e), the performance model prediction is used, taking into account the observed kernel benchmarks to produce an execution time, as explained in Section 6.1.

An execution time for a CPU and GPU implementation of XD-GRASP, (c), is calculated from the sum of the NUFFT time, using the GPU benchmarks, and the non-NUFFT time observed from an observation of CPU-only XD-GRASP measured in implementation (b).

[3]This path can be between state elements, inputs and outputs.

6.7 Bridging the Gap

The performance difference between the proposed system and a system capable of real-time reconstruction is one that can be closed. As mentioned, the acceleration of the NUFFT would be one way in which the total reconstruction time could be dramatically reduced. In terms of hardware, this would not require any changes to the system, since the GPU used is significantly under-utilised (less than 40% utilisation at its peak). Thus, rather than hardware changes, software changes would unlock the full potential of the system.

However, since the power demands of CPUs and GPUs are larger than that of an FPGA [31], it may be desirable to transfer all of the computation onto an FPGA, for uses when a low-power solution is required (such as portable reconstruction). In this case, the FPGA would likely require more hardware resources, mainly B/URAM, as this is became a more scarce resource in the current system.

These results support that real-time 4D MRI reconstruction is possible, should the NUFFT be accelerated, whilst maintaining the image accuracy when compared to the original XD-GRASP source. The work completed pushes towards the theoretical speed-up limit provided by Armdahl's Law.

7 Conclusion and Future Work

This report presents the modelling and optimisation approach for accelerating XD-GRASP using an FPGA and, further, a GPU. The use of FPGAs have successfully reduced the execution time of XD-GRASP and consequently provided a framework for the NUFFT to be optimised for, in this use case. The speed-up reaches 4.05x, when compared to the original MATLAB implementation, and 1.5384x over a C++ implementation, which lies close to the limit posed by Armdahl's Law, 1.5393. Likewise, the addition of a GPU achieves speed-ups of 6.63x (against MATLAB) and 1.8843x (against a CPU/GPU implementation) approaching the limit of 1.8862x. The work alleviates the computation burden of the processes of XD-GRASP, excluding NUFFT, by performing them in parallel with the largest bottleneck of the algorithm.

7.1 Future Work

This project demonstrates the potential for real-time MRI reconstruction, made possible by the use of an FPGA. To close the gap between the current and desired reconstruction time, the ideas in this subsection should be investigated.

7.1.1 NUFFT Acceleration

The current system does not achieve an execution time close enough to real-time, missing the real-time reconstruction target, but such performance can be obtained from the system should the NUFFT be accelerated. As modelled in Sections 3 and confirmed by the results in Section 6, the NUFFT is the main bottleneck for the current system.

Accelerating the NUFFT will have a significant, direct impact on the execution time of full reconstruction, since the NUFFT is such a large bottleneck as a result of the work done. As described, the performance model will be valid for a NUFFT acceleration of up to 136x, meaning a new NUFFT implementation can be evaluated ahead of its use in the system.

7.1.2 Pipelining Slices

In the same way that, on a micro level, the computation performed by kernel is pipelined, the reconstruction of 2D images can be pipelined on a macro level. Thus, a 4D reconstruction can be achieved in quicker time, rather than sequentially reconstructing 2D images for the respiratory phases as the current system does. In addition, this would better utilise the hardware, as multiple kernels could operate at a given time.

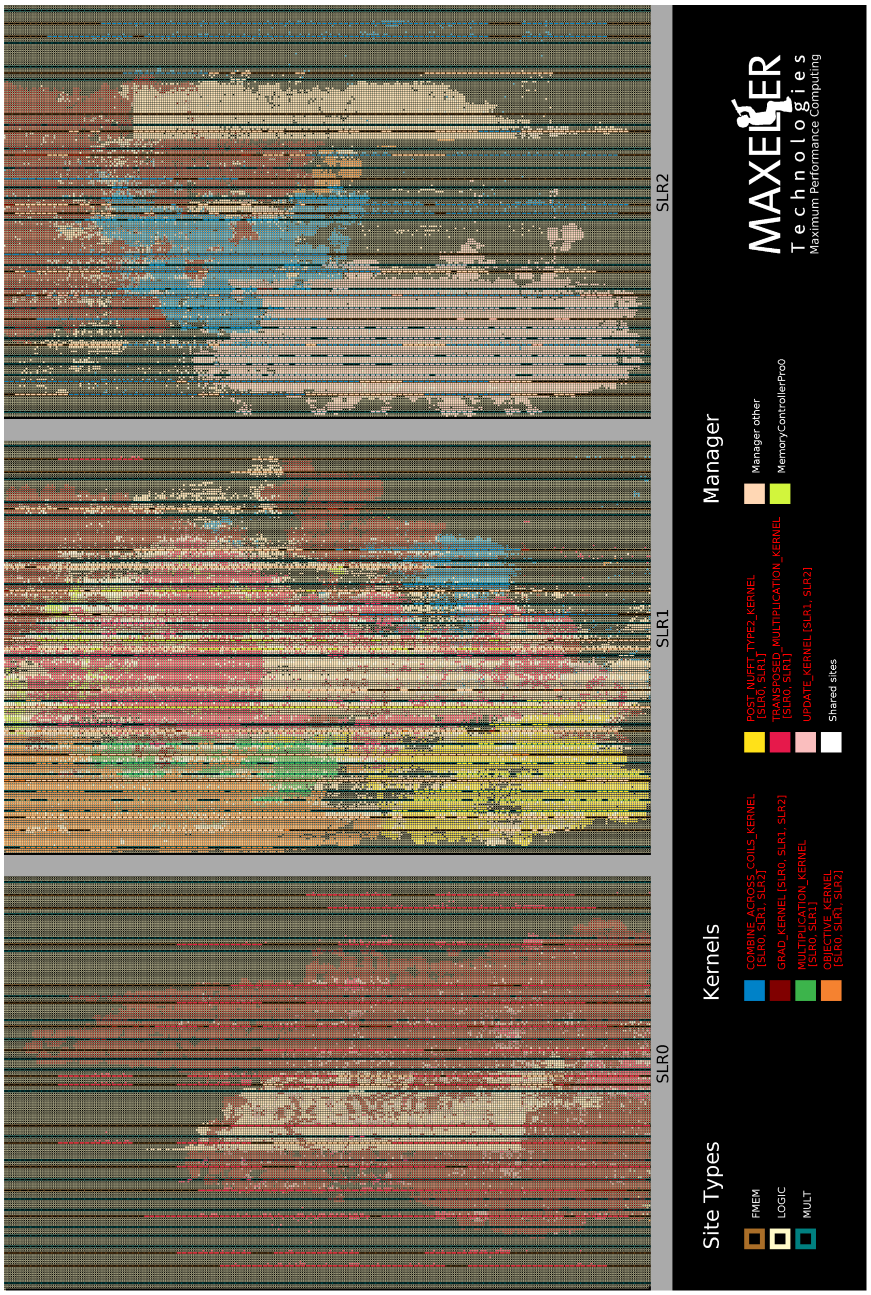
To make this reconstruction more user-friendly, the system could output the initial reconstruction as soon as possible, for preliminary viewing by the doctors, whilst the iterative improvements are made to the reconstruction (updating the view after each iteration).

7.1.3 Further FPGA Optimisation

There are cases where the FPGA functionality is not the most efficient it could be, as mentioned in the report. These can be addressed to maximise the performance of the system, but do not yield as much performance benefit as the previous future work suggestions do.

Appendices

A Aerial View of the FPGA Implementation



SLR2
SLR1
SLR0
MAXELER Technologies
Maximum Performance Computing
Site Types
FMEM
LOGIC
MULT
Kernels
COMBINE_ACROSS_COILS_KERNEL [SLR0, SLR1, SLR2]
GRAD_KERNEL [SLR0, SLR1, SLR2]
MULTIPLICATION_KERNEL [SLR0, SLR1]
OBJECTIVE_KERNEL [SLR0, SLR1, SLR2]
POST_NUFFT_TYPE2_KERNEL [SLR0, SLR1]
TRANSPOSED_MULTIPLICATION_KERNEL [SLR0, SLR1]
UPDATE_KERNEL [SLR1, SLR2]
Shared sites
Manager
Manager other
MemoryControllerPro0

B Further Kernel Details

The kernels not covered in Section 5.1 are discussed below.

B.1 `GradKernel`

This kernel performs the the final step of calculating the gradient during the conjugate gradient method, for one respiratory phase. Simultaneously, it computes the sum of the element-wise multiplication of the coefficients of `res` and their conjugates (the dot product of `res` and its conjugate).

In the cases where `x == xPrev` or `x == xNext`, the input denoting the respiratory phase, `r`, ensures the correct calculation, which does not make use of the equivalent input (i.e. `xPrev` and `xNext` respectively).

$$\begin{aligned}
\texttt{res}_i &= \texttt{L2Grad}_i + \texttt{TVWeight\_dim1} \cdot v \\
v &= \begin{cases} -v_2 & \texttt{r === 1} \\ v_1 & \texttt{r ===}\ ntres \\ v_1 - v_2 & \text{otherwise} \end{cases} \\
v_1 &= f(\texttt{x}_i - \texttt{xPrev}_i) \\
v_2 &= f(\texttt{xNext}_i - \texttt{x}_i) \\
f(x) &= \frac{x}{\sqrt{x \cdot \bar{x} + \texttt{l1Smooth}}} \\
\texttt{conjMultRes} &= \sum_{i=0}^{nPixels} |\texttt{res}_i \cdot \bar{\texttt{res}}_i|
\end{aligned}$$

- **Input:**
  - `L2Grad`: The output of a prior `CombineAcrossCoilsKernel` execution, for respiratory phase r
  - `x`: The reconstruction for respiratory phase r
  - `xPrev`: The reconstruction for respiratory phase $r_{Prev} = \max(0, r-1)$
  - `xNext`: The reconstruction for respiratory phase $r_{Next} = \min(r+1, MAX\_RESP\_PHASE)$

- `r`: The index of the current respiratory phase ($= r$)
- `l1Smooth`: A conjugate gradient smoothing parameter
- `TVWeight_dim1`: A conjugate gradient parameter for the weight of the temporal variance in calculation

• **Output:**

- `res`: The gradient for a respiratory phase
- `conjMultRes`: The dot product of `res` and its conjugate

• **Hardware requirements:**

- DSPs: $42 \times$ `pFactor`
- FMem: 0

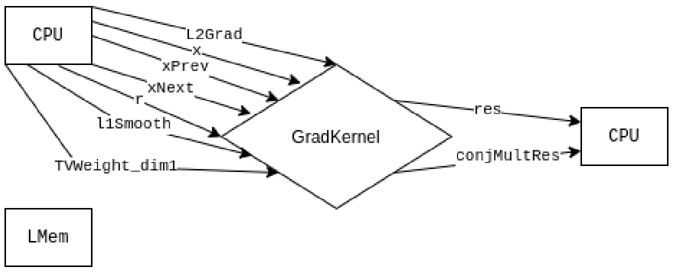


Figure 22: The dataflow of `GradKernel`.

B.2 `MultiplicationKernel`

This kernel simply multiplies its inputs to produce the input for the type 1 NUFFT which produces the initial reconstruction.

$$\texttt{out}_i = \texttt{kdatau}_i \cdot \texttt{wu}_i$$

• **Input:**

- `kdatau`: As stated above, for one respiratory phase and coil only
- `wu`: As stated above, for one respiratory phase only

- **Output:**
  - `out`: The element-wise multiplication of the inputs
- **Hardware requirements:**
  - DSPs: $6 \times$ `pFactor`
  - FMem: 0

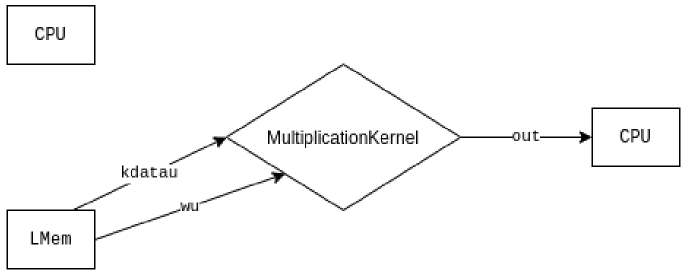


Figure 23: The dataflow of `MultiplicationKernel`.

B.3 `ObjectiveKernel`

As the final step of the objective function in backtracking line search, this kernel produces a single score for the output of the type 2 NUFFT, for one respiratory phase and coil.

$$\texttt{out} = \sum_{i=0}^{nSamples} z_i \cdot \bar{z}_i$$
$$z_i = \texttt{x}_i \cdot \texttt{wu}_i - \texttt{kdatau}_i$$

- **Input:**
  - `x`: The output of a type 2 NUFFT for a respiratory phase and coil
  - `kdatau`: As stated above, for one respiratory phase and coil only
  - `wu`: As stated above, for one respiratory phase only
- **Output:**
  - `out`: An objective cost for the input

- **Hardware requirements:**
  - DSPs: $12 \times$ `pFactor`
  - FMem: 0

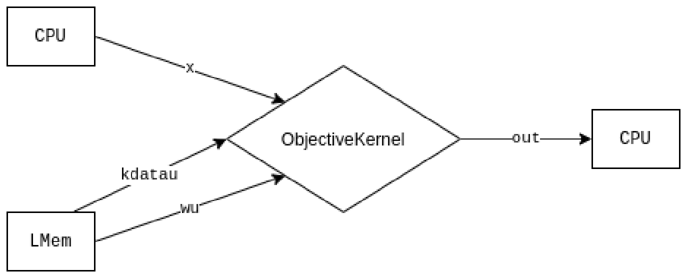


Figure 24: The dataflow of `ObjectiveKernel`.

B.4 `PostNUFFTType2Kernel`

During the computation of the gradient, this kernel is used to process the output of the type 2 NUFFT.

$$\texttt{out}_i = \left(\texttt{x}_i \cdot \texttt{wu}_i - \texttt{kdatau}_i\right) \cdot \texttt{wu}_i$$

- **Input:**
  - `x`: The output of a type 2 NUFFT for a respiratory phase and coil
  - `kdatau`: As stated above, for one respiratory phase and coil only
  - `wu`: As stated above, for one respiratory phase only
- **Output:**
  - `out`: Element-wise results for the input
- **Hardware requirements:**
  - DSPs: $12 \times$ `pFactor`
  - FMem: 0

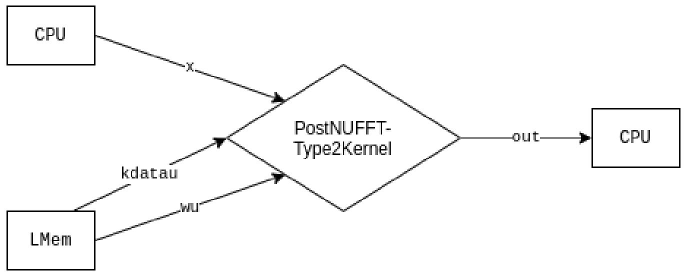


Figure 25: The dataflow of `PostNUFFTType2Kernel`.

B.5 `UpdateKernel`

Element-wise, this kernel produces the sum of the element of `a`, scaled by `scaleA`, and `b`, scaled by `scaleB`. Further, the sum of element-wise multiplications between `a` and the result, `out`, is output.

The name for this kernel derives from its most common use, updating the reconstruction using the gradient. However, the uses of this kernel go beyond this.

$$\texttt{out}_i = \texttt{a}_i \cdot \texttt{scaleA} + \texttt{b}_i \cdot \texttt{scaleB}$$

$$\texttt{conjMultRes} = \sum_{i=0}^{nPixels} \texttt{out}_i \cdot \bar{\texttt{a}_i}$$

- **Input:**
  - `a`: A vector
  - `scaleA`: A scalar
  - `b`: A vector
  - `scaleB`: A scalar
- **Output:**
  - `out`: The element-wise addition of the scaled inputs
  - `conjMultRes`: The dot product of `res` and `a`
- **Hardware requirements:**

- DSPs: 24 × `pFactor`
- FMem: 0

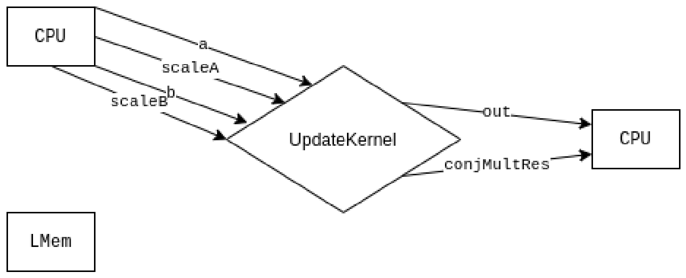


Figure 26: The dataflow of `UpdateKernel`.

References

[1] YouGov;. Date accessed: 17/01/2020. Available from: https://yougov.co.uk/topics/lifestyle/articles-reports/2011/08/15/cancer-britons-most-feared-disease.

[2] UK CR;. Date accessed: 17/01/2020. Available from: https://www.cancerresearchuk.org/health-professional/cancer-statistics/incidence.

[3] Feng L, Axel L, Chandarana H, Block KT, Sodickson DK, Otazo R. XD-GRASP: Golden-angle radial MRI with reconstruction of extra motion-state dimensions using compressed sensing. Magnetic Resonance in Medicine. 2016;75(2):775–788. ID: TN_wj10.1002/mrm.25665.

[4] Damadian R. Tumor detection by nuclear magnetic resonance. Science (New York, NY). 1971;171(3976):1151. ID: TN_medline5544870.

[5] Ramani R. Functional MRI : basic principles and emerging clinical applications in anesthesiology and the neurological sciences. Oxford University Press; 2019. ID: 44IMP_ALMA_DS51115923850001591; Includes bibliographical references and index.

[6] Mezrich R. A perspective on K-space. Radiology. 1995 May;195(2):297–315. Available from: https://www.ncbi.nlm.nih.gov/pubmed/7724743.

[7] Gallagher TA, Nemeth AJ, Hacein-Bey L. An introduction to the Fourier transform: relationship to MRI. AJRAmerican journal of roentgenology. 2008;190(5):1396. ID: TN_medline18430861.

[8] Lustig M, Donoho D, Pauly JM. Sparse MRI: The application of compressed sensing for rapid MR imaging. Magnetic Resonance in Medicine. 2007;58(6):1182–1195.

[9] Ruiz-Antolin D, Townsend A. A nonuniform fast Fourier transform based on low rank approximation; 2017. 1701.04492.

[10] Frigo M, Johnson SG. FFTW;. Date accessed: 17/01/2020. Available from: http://www.fftw.org/.

[11] Barnett A, Magland J. FINUFFT;. Date accessed: 17/01/2020. Available from: `https://finufft.readthedocs.io/en/latest/index.html`.

[12] Barnett A, Magland J. FINUFFT: Usages;. Date accessed: 17/01/2020. Available from: `https://finufft.readthedocs.io/en/latest/usage.html`.

[13] Schwartzl A, Knoll F. gpuNUFFT;. Date accessed: 17/05/2020. Available from: `https://github.com/andyschwarzl/gpuNUFFT`.

[14] NVidia. CUDA;. Date accessed: 24/05/2020. Available from: `https://developer.nvidia.com/cuda-zone`.

[15] Weisstein EW. L2-Norm;. Date accessed: 17/01/2020. Available from: `http://mathworld.wolfram.com/L2-Norm.html`.

[16] Cauchy A. Méthode générale pour la résolution des systemes d'équations simultanées. Comp Rend Sci Paris. 1847;25.1847:536–538.

[17] Gradient descent;. Date accessed: 17/01/2020. Available from: `https://en.wikipedia.org/w/index.php?title=Gradient_descent`.

[18] Chen X. Inverse Scattering Problems of Small Scatterers; 2018. ID: TN_wilbooks10.1002/9781119311997.ch4.

[19] Dennis JE. Numerical methods for unconstrained optimization and nonlinear equations; 1996. ID: 44IMP_ALMA_DS5153880780001591; Includes bibliographical references (p. 364-370) and indexes.

[20] Armijo L. Minimization of functions having Lipschitz continuous first partial derivatives. Pacific JMath. 1966;16(1):1–3. ID: TN_euclideuclid.pjm/1102995080.

[21] Donoho DL. Compressed sensing. IEEE Transactions on Information Theory. 2006;52(4):1289–1306. ID: TN_ieee_s1614066.

[22] Deshmane A, Gulani V, Griswold MA, Seiberlich N. Parallel MR imaging. Journal of Magnetic Resonance Imaging. 2012;36(1):55–72. ID: TN_wj10.1002/jmri.23639.

[23] Feng L, Grimm R, Block KT, Chandarana H, Kim S, Xu J, et al. Golden-angle radial sparse parallel MRI: Combination of compressed sensing, parallel imaging, and golden-angle radial sampling for fast and flexible dynamic volumetric MRI. Magnetic Resonance in Medicine. 2014;72(3):707–717. ID: TN_wj10.1002/mrm.24980.

[24] Pang J, Sharif B, Fan Z, Bi X, Arsanjani R, Berman DS, et al. ECG and navigator-free four-dimensional whole-heart coronary MRA for simultaneous visualization of cardiac anatomy and function. Magnetic Resonance in Medicine. 2014;72(5):1208–1217.

[25] Palubinskas G. Image similarity/distance measures: what is really behind MSE and SSIM? International Journal of Image and Data Fusion. 2017;8(1):32–53. ID: TN_informaworld_s10_1080_19479832_2016_1273259.

[26] Wang Z, Bovik AC. Mean squared error: Love it or leave it? A new look at Signal Fidelity Measures. IEEE Signal Processing Magazine. 2009;26(1):98–117. ID: TN_ieee_s4775883.

[27] Wang Z, Bovik AC. A universal image quality index. IEEE Signal Processing Letters. 2002;9(3):81–84. ID: TN_ieee_s995823.

[28] Xilinx VU9P Manual;. Available from: `https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf`.

[29] Maxeler. Maxeler Website;. Date accessed: 02/06/2020. Available from: `http://maxeler.com/`.

[30] Wang Y, He Y, Shan Y, Wu T, Wu D, Yang H. Hardware computing for brain network analysis. In: 2nd Asia Symposium on Quality Electronic Design (ASQED); 2010. p. 219–222.

[31] Siddiqui MF, Reza AW, Shafique A, Omer H, Kanesan J. FPGA implementation of real-time SENSE reconstruction using pre-scan and Emaps sensitivities. Magnetic Resonance Imaging. 2017;44:82 – 91. Available from: `http://www.sciencedirect.com/science/article/pii/S0730725X17301674`.

[32] Pruessmann KP, Weiger M, Scheidegger MB, Boesiger P. SENSE: Sensitivity encoding for fast MRI. Magnetic Resonance in Medicine. 1999;42(5):952–962. ID: TN_wjAID-MRM16¿3.0.CO; ID: 2-S.

[33] Li L, Wyrwicz AM. Design of an MR image processing module on an FPGA chip. Journal of Magnetic Resonance. 2015;255:51–58.

[34] Stone SS, Haldar JP, Tsao SC, Hwu WMW, Sutton BP, Liang ZP. Accelerating advanced MRI reconstructions on GPUs. Journal of Parallel and Distributed Computing. 2008;68(10):1307–1318.

[35] Feng L, Axel L, Chandarana H, Block KT, Sodickson DK, Otazo R. XD-GRASP Demo Code (MATLAB);. Date accessed: 24/05/20. Available from: `https://cai2r.net/resources/software/xd-grasp-matlab-code`.

[36] ICR. The Institute of Cancer Research Website;. Date accessed: 24/05/2020. Available from: `https://www.icr.ac.uk/`.

[37] Wikipedia. Wikipedia - Armdahl's Law;. Date accessed: 02/06/2020. Available from: `https://en.wikipedia.org/wiki/Amdahl%27s_law`.