Sparse Image Approximation in 3-Dimensions with GPU Utilisation

Daniel Whitehouse

Supervised By:

Dr Laura Rebollo-Neira

and

Dr George Vogiatzis

April 28, 2017

A dissertation submitted to the School of Engineering and Applied Sciences, Aston University in partial fulfilment of the requirements for the degree BSc (Hons) Computer Science and Mathematics.

1 Abstract

In the area of signal processing, Matching Pursuit and its adaptations have already been proven to be an effective approach to Sparse Approximation of Images. With the continuous development of GPU performance, there is a case to make a contribution in this field. Subsequently, revised algorithms dedicated to 3 dimensions named 'Self-Projected Matching Pursuit' (SPMP3D) and 'Orthogonal Matching Pursuit' (OMP3D) have been implemented in this report. Since these algorithms are novel, their design and the approach towards further development have been discussed in detail. To improve the current MATLAB prototypes, they have been implemented in C++ with optimisation through BLAS; achieving a significant performance gain. The construction of a GPU-accelerated command line application for calculating the sparsity ratio of a given image using the MP3D algorithm has been produced. Additionally, an extension to fully support the SPMP3D algorithm has been outlined. A study of the sparsity gain in colour images, by using the algorithms dedicated to the simultaneous processing of 3 colour channels, has been conducted. The developed applications were also reviewed based on their performance compared to the MATLAB prototypes and were found to be a significant improvement. These applications have been produced as open source and have been made publicly available through GitHub for future development.

Project Definition

Student's name – Daniel Whitehouse
Course – Computer Science and Mathematics BSc
Project title – Self-Projected Matching Pursuit for Objects in 3-Dimensions
What is the project about? It will develop a tool that is capable of applying the Self-Projected Matching Pursuit (SPMP) algorithm to an image to find the sparse representation of that image, in particular, it aims to apply this technique to images/objects in 3 dimensions. Matching pursuit are greedy algorithms, and image processing is by nature compute-intensive and time consuming, so the project will look at the utilisation of GPU to optimise the procedure.
What is the project deliverable? The project deliverable will be a tool for the scientific community, written in C++ which will be optimised by using NVIDIA's CUDA platform which would allow the execution of parallel calculations on the GPU. There will be an investigation into the results which would look at the efficiency regarding computation time with the use of a GPU, also effectiveness by measuring the sparsity of the approximation with the original image. Support for the tool to be called from MATLAB by using MEX may also be part of the deliverable.
What is original about this project? To the best of my knowledge, there has not been an implementation of a greedy matching pursuit algorithm such as SPMP for use on images of 3 dimensions which is the goal of this project. A C++ implementation (which is a literal translation from MATLAB to C++) already exists for sparse approximation of 2D images and uses MEX so that it is executable from MATLAB; this implementation does not use the GPU for acceleration. This project will aim to build upon or extend the existing implementation to support GPU & 3D objects.
Timetable showing main stages in work plan Start: 28/09/16 End: 21/10/16 - Background research into existing implementations and most suitable programming environment Start: 21/10/16 End: 7/11/16 - Establish thorough understanding of SPMP algorithm and working prototype implementation for use in (2D) Start: 07/11/16 End: 14/11/16 - Elicit required classes and design the program so that it can be called standalone Start: 14/11/16 End: 28/11/16 - Implement CUDA methods for matrix operations used in the matching pursuit algorithm Start: 28/11/16 End: 12/12/16 – Incorporate CUDA operations in the 2D algorithm Start: 12/12/16 End: 07/02/17 - Create a prototype algorithm to support 3D objects. Also, a prototype to load 3D objects into the tool Start: 07/02/17 End: 20/02/17 - Extend the program to use MEX for use with MATLAB Start: 20/02/17 End: 27/02/17 – Code cleanup and exception handling Start: 27/02/17 End: 06/03/17 - Investigation and analysis into the results of the tool Start: 06/03/17 End: 07/04/17 – Create and complete the final report

Student's signature  *Date* 28/04/17

Supervisor's signature  *Date* 28/04/17

Ethics form for student projects

SEAS group: Computer Science

Project title: Self Projected Matching Pursuit for Objects in 3-Dimensions

Supervisor name and email: Dr. George Vogiatzis - vogiatzg@aston.ac.uk

Ethics questions

Please answer Yes or No to each of the following four questions:

1 - Does the project involve participants selected because of their links with the NHS/clinical practice or because of their professional roles within the NHS/clinical practice, or does the research take place within the NHS/clinical practice, or involve the use of video footage or other materials concerning patients involved in any kind of clinical practice? ***No***

2 - Does the project involve any i) clinical procedures or ii) physical intervention or iii) penetration of the participant's body or iv) prescription of compounds additional to normal diet or other dietary manipulation/supplementation or v) collection of bodily secretions or vi) involve human tissue which comes within the Human Tissue Act? (eg surgical operations; taking body samples including blood and DNA; exposure to ionizing or other radiation; exposure to sound light or radio waves; psychophysiological procedures such as fMRI, MEG, TMS, EEG, ECG, exercise and stress procedures; administration of any chemical substances)? ***No***

3 - Having reflected upon the ethical implications of the project and/or its potential findings, do you believe that that the research could be a matter of public controversy or have a negative impact on the reputation/standing of Aston University? ***No***

4 - Does the project involve interaction with or the observation of human beings, either directly or remotely (eg via CCTV or internet), including surveys, questionnaires, interviews, blogs, etc? *Answer "no" if you are only asking adults to rate or review a product that has no upsetting or controversial content, you are not requesting any personal information, and the adults are Aston employees, students, or your own* friends. ***No***

Student's signature: 

Supervisor's signature: 

Please submit this form as part of your Term 1 Progress Report. If any of the answers are "yes", you will need to complete an online application for ethics approval, which can be found at https://www.ethics.aston.ac.uk *. You can log in with your usual Aston user name and password.*

Acknowledgements

I would firstly like to express my great appreciation to Dr Laura Rebollo-Neira for her continuous and patient guidance throughout this project, and for her confidence in approaching me to tackle the proposal to begin with.

I would also like to express my deep gratitude to Dr George Vogiatzis for his technical advice and valuable assistance whom which this project would not have been a success.

I would like to offer my special thanks to the Mathematics Department for providing the specialised equipment and workspace which is usually reserved for PhD. Students.

Finally, I wish to extend my thanks to my friends and family for their enduring support and encouragement throughout my study.

List of contents

2 Introduction

In a world where the quality and volume of data is ever increasing, the need for effective compression is imperative to drastically reduce storage requirements (King, 2013). The scope of this data, includes the diverse and rapid increase in searchable image data spanning geographically disparate locations (Datta, Joshi, Li, & Wang, 2008). The paper "Using Compression Techniques to Streamline Image and Video Storage and Retrieval" (Krishnan, 2014) highlights the drawbacks of using uncompressed multimedia, regarding storage, bandwidth throttling and the financial cost of hosting such inefficient files.

Fortunately this is not the case, and image compression is already widely used, in particular on the internet where the three most common image formats are PNG, GIF and JPEG (Michigan Library, 2017) – each has their advantages and shortcomings which are discussed in Section **3.1**. As with many other image compression techniques, a common initial step is mapping the image onto transformed space where there can be a reduction in the number of elements which make up the image. For instance, JPEG as such uses the Discrete Cosine Transformation (DCT) for this step. However, improvements in the sparsity of these transformations can be increased by using a set of sparse approximation methods called 'pursuit strategies'.

This report focuses on two adapted pursuit strategies: Orthogonal Matching Pursuit (OMP) (Pati, Rezaiifar, & Krishnaprasad, 1993), its extensions to 2D, termed OMP2D (Rebollo-Neira, Bowley, Constantinides, & Plastino, 2012) and Self-Projected Matching Pursuit (SPMP) (Rebollo-Neira & J.Bowley, 2013) which will be explained in Section **4 – Preparation**. The notable highlight in this report is the novel use of a 3D separable dictionary i.e. a 3D dictionary which is the tensor product of three 1D dictionaries. The intention of having these three dictionaries is to see if the 3D processing can outperform previous sparse image approximation techniques for data reduction. Past reports have already shown that pursuit strategies show increasing sparsity gain compared to other transformations like the DCT and Discrete Wavelet Transform (DWT) which JPEG and JPEG2000 use respectively (Rebollo-Neira et al., 2012). The background section will also cover in depth, existing strategies of Sparse Image Approximation as well as limitations found in other papers.

The **Preparation** section of the paper then focuses in more detail on the specific pursuit strategies used in this application, and defines the various objective measures that are going to be used while evaluating the algorithms. This section also provides a background for the three technologies that were studied to build the applications.

The report then moves on to the **Deliverable** section, which examines how the discussed Sparse Image Approximation techniques were implemented using the aforesaid technologies and explains the approach taken to realise the design. Since these computationally intensive routines currently only exist in MATLAB, a library of C++ extension routines have been developed as well as an application using NVidia's Compute Unified Device Architecture (CUDA) framework, in an attempt to accelerate the performance. The C++ implementation, CUDA application and numerical experiments concerning multi-channel images should be considered as the deliverables of this report.

A systematic **Evaluation** is then conducted to gather useful findings of the effectiveness of approximating all colour channels simultaneously, as opposed to separately, which has been the way in previous work. A performance evaluation is also undertaken to compare the efficiency of both the provided C++ extension routines and the CUDA application to the MATLAB counterpart.

The report is then summarised, and a reflection on the work done is assumed in the **Conclusion** section. Future works which could stem from this project are then discussed.

2.1 Statement of Contributions

- Production of C++ MATLAB executables for SPMP3D and OMP3D which significantly increased performance compared to the stand alone MATALB implementation.
- A study into the feasibility of using the GPU and fast-access memory for the SPMP3D routine.
- Development of a CUDA enabled command line application for calculating the sparsity ratio of a given image using the MP3D algorithm. In addition, an extension to fully support the SPMP3D algorithm has been outlined.
- An original investigation into the benefit of approximating in 3-dimensions by approximating colour channels simultaneously instead of approximating colour channels independently.
- A meaningful investigation into the performance gain from using C++ and GPU acceleration.
- Creation of a Project Webpage hosted on GitHub pages for public use once submitted and approved. This will allow future evolution of the deliverables.
    - Can be found at : https://dannyxd11.github.io/SIA_3D/docs/

3 Background

This section briefly discusses current image formats and the notion of sparse image approximation. Sections **3.3** and **3.4** are drawn from "Notes on Greedy Strategies for Sparse 3D Image Approximation" (Rebollo-Neira L. , 2017). A review of some previous work in this field is also considered in Section **3.5**.

3.1 Image Formats

As mentioned, PNG, GIF, and JPEG are the most common image formats used, but each have their own advantages and disadvantages, as now discussed. The Graphics Interchange Format (GIF) is lossless, can be animated and can store transparency, however, it is limited by the fact it can only store 256 different colours in its colour palette.

Moreover, Portable Network Graphic (PNG) was designed to replace GIF – it is also lossless, capable of using 16 million colours (true colour), with transparency, but can fall short with the achieved compression ratio of high-quality images since no data is discarded in the process.

The Joint Photographers Expert Group (JPEG) format which is lossy, so can often create artefacts and blur points in an image (Bourque, 2014). Nevertheless, JPEG can achieve very high compression ratios whilst maintaining good image quality and it is for this reason the JPEG format is predominantly used for photographs (Kaur & Choudhary, 2016). There is also a JPEG2000 format which improves the compression ratio as well as some of JPEGs weaknesses, for example, it is more error resilient, extendable and offers support for lossless compression (Joint Photographers Expert Group, 2000).

It is also important to mention there are four other steps involved in JPEG compression. Firstly, the image is downsampled so only some of the elements are extracted and used. Then the DCT is applied with an output of 8x8 matrix of coefficients. These coefficients are quantized by dividing by empirically derived quantization matrices. At this point the matrix is sparser than the original image. The upper left section of the matrix will contain most of the elements which represents the lower frequencies, this is the desirable property known as energy compaction. After this step, two common entropy encoding techniques are applied, firstly run length encoding on the quantized matrix in a zig-zag fashion starting in the upper left corner. This is effective since it is possible for majority of the bottom-right of the matrix to be zero. The final step is applying Huffman encoding.

This report discusses sparse representation in relation to data reduction which is the first step in a lossy transform compression scheme.

3.2 Mathematical Notation

Let us first set the scene by introducing the mathematical notation that will be used throughout this paper. Following convention, let $\mathbb{R}$ and $\mathbb{N}$ represent the sets of real and natural numbers. Boldface letters indicate Euclidean vectors or matrices, whilst typical mathematical font represents elements. For instance, $\mathbf{d} \in \mathbb{R}^N$ is a vector of components $d(i)$, with $i = 1, \ldots, N$ and $\mathbf{I} \in \mathbb{R}^{N_x \times N_y \times N_z}$ is a 3D array of elements $I(i,j,k),\ i = 1, \ldots, N_x,\ j = 1, \ldots, N_y$ and $k = 1, \ldots, N_z$. Furthermore, $\mathbf{I}^K \in \mathbb{R}^{N_x \times N_y}$ for $K = 1, \ldots, N_z$ represents the K^{th} plane of $\mathbf{I}$ such that $\mathbf{I}^K(i,j) = \mathbf{I}(\mathrm{i}, \mathrm{j}, \mathrm{k})$.

The inner product between 3D arrays $\mathbf{A} \in \mathbb{R}^{N_x \times N_y \times N_z}$ and $\mathbf{I} \in \mathbb{R}^{N_x \times N_y \times N_z}$ is given by:

$$\langle \mathbf{A}, \mathbf{I} \rangle_{3D} = \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} \sum_{k=1}^{N_z} A(i,j,k) I(i,j,k). \tag{1}$$

For $\mathbf{A} \in \mathbb{R}^{N_x \times N_y \times N_z}$ defined by the tensor product such that $\mathbf{A} = \boldsymbol{d}^x \otimes \boldsymbol{d}^y \otimes \boldsymbol{d}^z$, with $\boldsymbol{d}^x \in \mathbb{R}^{N_x}$, $\boldsymbol{d}^y \in \mathbb{R}^{N_y}$ and $\boldsymbol{d}^z \in \mathbb{R}^{N_z}$, we further have:

$$\langle \mathbf{A}, \mathbf{I} \rangle_{3D} = \sum_{k=1}^{N_z} \langle \mathbf{d}^x, \mathbf{I}^k\, \mathbf{d}^y \rangle\, d^z(k) = \langle \mathbf{p}, \mathbf{d}^z \rangle, \tag{2}$$

where for each value of k, $\mathbf{I}^k\, \mathbf{d}^y$ is given by the usual matrix-vector multiplication and the components of $\mathbf{p} \in \mathbb{R}^{N_z}$ are given by $p(k) = \langle \mathbf{d}^x, \mathbf{I}^k\, \mathbf{d}^y \rangle,\ k = 1, \ldots, N_z$. Also note $\langle \mathbf{p}, \boldsymbol{d}^z \rangle$ is calculated using the standard Euclidean inner product:

$$\langle \mathbf{p}, \boldsymbol{d}^z \rangle = \sum_{k=1}^{N_z} p(k)\, d^z(k). \tag{3}$$

3.3 Sparse Image Approximation

Given an image, in which the intensity of pixels is expressed in the form $\mathbf{I} \in \mathbb{R}^{N_x \times N_y \times N_z}$, we can approximate the image by linear decomposition such that:

$$\mathbf{I}^k = \sum_{n=1}^{k} c(n) \mathbf{D}_{\ell_n}, \tag{4}$$

where each component in $\boldsymbol{c}$ is a scalar and each $\mathbf{D}_{\ell_k} \in \mathbb{R}^{N_x \times N_y \times N_z}$ is to be selected from a set $\mathcal{D} = \{\mathbf{D}_n\}_{n=1}^{M}$ termed a dictionary.

A sparse approximation of an image, $\mathbf{I} \in \mathbb{R}^{N_x \times N_y \times N_z}$, is one in the form of (4) such that the number of elements k is significantly smaller than $N = N_x N_y N_z$. The terms in the decomposition are taken from a large redundant dictionary from where the elements $\mathbf{D}_{\ell_n}$, called 'atoms' are chosen based on an optimality criterion.

The aim is to find the most sparse decomposition of an image, that is, the atomic decomposition for which the number of elements k is at a minimum. However, this requires an undesirable exhaustive

search so a different approach is taken. Instead of searching for the most sparse solution, a solution that is satisfactory, i.e. one that the number k terms in the decomposition is considerably smaller than the image dimensions *N*.

Consider a 3D dictionary $\mathcal{D} = \{\mathbf{D}_i \in \mathbb{R}^{N_x \times N_y \times N_z}\}_{i=1}^{M}$ is obtained by the tensor product of three 1D dictionaries such that $\mathcal{D} = \mathcal{D}^{\boldsymbol{x}} \otimes \mathcal{D}^{\boldsymbol{y}} \otimes \mathcal{D}^{\boldsymbol{z}}$ where $\mathcal{D}^x = \{\mathbf{d}_n^x \in \mathbb{R}^{N_x}\}_{n=1}^{M_x}$, $\mathcal{D}^y = \{\mathbf{d}_n^y \in \mathbb{R}^{N_y}\}_{n=1}^{M_y}$ and $\mathcal{D}^z = \{\mathbf{d}_n^z \in \mathbb{R}^{N_z}\}_{n=1}^{M_z}$ with $M_x M_y M_z = M$. Given a 3D array $\mathbf{I} \in \mathbb{R}^{N_x \times N_y \times N_z}$ the aim is to approximate by atomic decomposition so it is in the form:

$$\mathbf{I}^k = \sum_{n=1}^{k} c^k(n) \mathbf{d}_{\ell_n^x}^x \otimes \mathbf{d}_{\ell_n^y}^y \otimes \mathbf{d}_{\ell_n^z}^z. \tag{5}$$

Each of the discussed methods for construction of approximations in the form (5) select atoms in steps over a number of iterations until some criterion, for our case the quality based on the PSNR (Section 4.1.3.1), is achieved. On the first iteration $k = 0$ and setting the initial residual $\mathbf{R}^0 = \mathbf{I}$, the algorithm at step $k + 1$ selects the indices ℓ_{k+1}^x, ℓ_{k+1}^y and ℓ_{k+1}^z by choosing the indices that maximise the absolute value of the inner product between the dictionary and the residual $\mathbf{R}^k$ as shown below:

$$\ell_{k+1}^{\boldsymbol{x}}, \ell_{k+1}^{\boldsymbol{y}}, \ell_{k+1}^{\boldsymbol{z}} = \underset{\substack{n=1\ldots M_x \\ s=1\ldots M_y \\ t=1\ldots M_z}}{\mathbf{argmax}} | \langle \mathbf{d}_n^x \otimes \mathbf{d}_s^y \otimes \mathbf{d}_t^z, \mathbf{R}^k \rangle_{3\mathrm{D}} |, \tag{6}$$

where the residual is given by $\mathbf{R}^k = \mathbf{I} - \mathbf{I}^k$. It is the difference in the ways the strategies determine the coefficients $c^k(n) = 1, \ldots, k$ that distinguishes them by name.

3.4 3D Matching Pursuit Algorithm

The original Matching Pursuit (MP) algorithm was first published in (Mallat et al., 1993) and is the simplest method of calculating the coefficients. However, an adapted version to handle 3D separable dictionaries is shown here (Rebollo-Neira L., 2017). The MP approach in 3D (MP3D) would simply calculate the coefficients in (5) as:

$$c(n)^k = \langle \mathbf{d}_{\ell_n^x}^x \otimes \mathbf{d}_{\ell_n^y}^y \otimes \mathbf{d}_{\ell_n^z}^z, \mathbf{R}^k \rangle_{3\mathrm{D}}. \tag{7}$$

The shortcoming of the MP algorithm is that it may choose atoms that are linearly dependent i.e. chosen atoms can be represented as a combination of other chosen atoms. Furthermore, the approximation is not optimal since the norm of the residual is not minimised at each step, consequently more atoms must be chosen, decreasing the resulting sparsity.

The deficiencies of the MP3D routine are overcome in the OMP routine described in Section **4.1.1**.

3.5 Previous Work

Since the project focuses on effective Sparse Image Approximation dedicated to 3D images, and to the best of my knowledge this has not been previously considered, the previous papers upon which this work is based are:

1. The seminal Matching Pursuit paper (Mallen et al, 1993) – This simple method has been tested here in 3D as a particular case for an improved algorithm.
2. The seminal Orthogonal Matching Pursuit paper (Pati et al, 1993). This approach has been studied within the specific implementation proposed in (Rebollo-Neira et al 2012). That implementations, which had been previously shown to produce superior sparsity results in comparison with the DCT and DWT, have been extended here to approximate 3D images. The advantages of this extension will be illustrated by the simultaneous approximation of multichannel images, in comparison with the independent approximation of each channel.
3. The Self-Projected Matching Pursuit paper (Rebollo-Neira et al 2013) has also been studied in detail, since its effective implementation in 3D is the goal of this project.

Another paper relating to the projects aim considers a GPU implementation of OMP for sparse 2D image approximation (Sattigeri, Thiagarajan, Ramamurthy, & Spanias, 2012). However, the method which is implemented is not dedicated to 2D images. It simply vectorises the image and applies OMP. Whilst the image is partitioned into blocks of 8 x 8, each block is transformed into a vector of length 64. Thus, the dictionary they use for the implementation is a relatively small one. It only consists of 1024 vectors of length 64, which implies a redundancy of $\frac{1024}{64} = 16$. The paper justifies this by needing to reduce to computational complexity and storage requirements of the OMP routine, due to the 'limited availability of fast access memory available to each thread to compute the sparse code'. The implementation of SPMP in this report does not have the same demanding storage requirements as OMP, it also benefits by having a higher redundancy of $\frac{40\times40\times15}{8\times8\times3} = 125$.

The inefficiency in the implementation outlined in (Sattigeri et al, 2012) is the storage requirements of the dictionary. Assuming the dictionary is stored in double precision, it requires 512kb of storage, thus it must be stored in global memory for all the threads to have access to it. This comes with increased memory latency when accessing components of the dictionary. However, the approach with three separable dictionaries discussed in this report is small enough to be stored on the GPUs shared memory, thus can take full advantage of fast access memory. The approach discussed in this report does not vectorise the block but rather approximates all three dimensions off the block together.

4 Preparation

In this section, the mathematics behind the aforementioned pursuit strategies, the architecture of the hardware and the three technologies that were used in the deliverable; C++, MATLAB with MEX and CUDA are discussed. The section begins by covering in detail the evolved MP algorithms used to achieve sparse image approximations discussed in Section **3.3**, namely Orthogonal Matching Pursuit (OMP) and the less memory demanding SPMP.

Since the author is unconversant with C++, MEX files and CUDA, the report has gone into more detail about their background, design and general use, considering a notable amount of time on the project was spent becoming familiar with these technologies. In the deliverable section we move away from the abstract use and further explain how these technologies were used specifically in this application.

4.1 Matching Pursuit Algorithms

4.1.1 3D Orthogonal Matching Pursuit (OMP3D)

For the OMP algorithm, an extension to OMP2D described in (Rebollo-Neira et al., 2012) is presented from (Rebollo-Neira L. , 2017) to accommodate the third dimension. It is established on the methods of biorthogonalization and Gram-Schmidt orthogonalization proposed in (Rebollo-Neira & Lowe, 2002). To ensure the coefficients $c^k(n)$, n = 1, ... , k used in (5) are optimal, the norm of the residual $\left\|\mathbf{R}^k\right\|_{3D}^2 = \langle \mathbf{R}^k, \mathbf{R}^k \rangle_{3D}$ at each iteration should be minimised. This is achieved by ensuring the atomic decomposition satisfies:

$$\mathbf{I}^k = \sum_{n=1}^{k} c^k(n) \mathbf{d}_{\ell_n^x}^x \otimes \mathbf{d}_{\ell_n^y}^y \otimes \mathbf{d}_{\ell_n^z}^z = \widehat{\mathrm{P}}_{\mathbb{V}_k} \mathbf{I}, \tag{8}$$

where $\widehat{\mathrm{P}}_{\mathbb{V}_k}$ is the orthogonal projection operator onto the space spanned by $\mathbb{V}_k = \mathrm{span}\{\mathbf{d}_{\ell_n^x}^x \otimes \mathbf{d}_{\ell_n^y}^y \otimes \mathbf{d}_{\ell_n^z}^z\}_{n=1}^k$. At each iteration, the residual error is calculated as $\mathbf{R}^k = \mathbf{I} - \widehat{\mathrm{P}}_{\mathbb{V}_k} \mathbf{I}^k$. The representation $\widehat{\mathrm{P}}_{\mathbb{V}_k} \mathbf{I}$ is of the form $\widehat{\mathrm{P}}_{\mathbb{V}_k} \mathbf{I} = \sum_{n=1}^{k} \mathbf{A}_n \langle \mathbf{B}_n^k, \mathbf{I} \rangle_{3D}$, where $\mathbf{A}_n \in \mathbb{R}^{N_x \times N_y \times N_z}$ is an array which contains the selected atoms $\mathbf{A}_n = \mathbf{d}_{\ell_n^x}^x \otimes \mathbf{d}_{\ell_n^y}^y \otimes \mathbf{d}_{\ell_n^z}^z$. The concomitant array $\mathbf{B}_n \in \mathbb{R}^{N_x \times N_y \times N_z}$, $n = 1,\ldots,$ k, satisfies the conditions:

i. $\langle \mathbf{A}_n, \mathbf{B}_m^k \rangle_{3D} = \delta_{n,m} = \begin{cases} 0 \; if \; n \neq m, \\ 1 \; if \; n = m. \end{cases}$

ii. $\mathbb{V}_k = \mathrm{span}\{\mathbf{B}_n^k\}_{n=1}^k$.

The required array **B** can be constructed at each iteration by extending the formula outlined in (Rebollo-Neira et al, 2002). As such:

$$\mathbf{B}_n^{k+1} = \mathbf{B}_n^k - \mathbf{B}_{k+1}^{k+1} \langle \mathbf{A}_{k+1}, \mathbf{B}_n^k \rangle_{3D}, \quad n = 1, \ldots, k\,, \tag{9}$$

where

$$\mathbf{B}_{k+1}^{k+1} = \frac{\mathbf{W}_{k+1}}{\|\mathbf{W}_{k+1}\|_{3D}^2}, \text{with } \mathbf{W}_1 = \mathbf{A}_1 \text{ and } \mathbf{W}_{k+1} = \mathbf{A}_{k+1} - \sum_{n=1}^{k} \frac{\mathbf{W}_n}{\|\mathbf{W}_n\|_{3D}^2} \langle \mathbf{W}_n, \mathbf{A}_{k+1} \rangle_{3D}. \quad (10)$$

When the arrays **B** are constructed as above, the coefficients for the iteration can then be obtained by the inner product:

$$c_n = \langle \mathbf{B}_n^k, \mathbf{I} \rangle_{3D}, n = 1, \dots, k\,. \quad (11)$$

Although efficient, the memory overhead of having to store the arrays **W** and **B** in OMP3D are too costly for some implementations, for instance on the GPU where low latency memory is a limited resource. For this reason, another pursuit strategy, SPMP3D is considered which eliminates the need to store the arrays $\mathbf{W}_n$ and $\mathbf{B}_n^k, n = 1, \dots, k$.

4.1.2 3D Self-Projected Matching Pursuit (SPMP3D)

The 2D implementation of SPMP was introduced in (Rebollo-Neira et al, 2012). As with the OMP2D algorithm the SPMP2D algorithm can easily be extended to support the third dimension by using the inner product defined in (1) or (2).

Suppose that at each iteration *k* the selection process has chosen the atoms labelled by the triple of indices $\{\ell_n^x, \ell_n^y, \ell_n^z\}_{n=1}^k$ and let $\tilde{\mathbf{I}}^k$ be the atomic decomposition

$$\tilde{\mathbf{I}}^k = \sum_{n=1}^{k} a(n) \mathbf{d}_{\ell_n^x}^x \otimes \mathbf{d}_{\ell_n^y}^y \otimes \mathbf{d}_{\ell_n^z}^z, \quad (12)$$

where *a(n)*, *n* = 1, ... , *k* are the coefficients. $\tilde{\mathbf{I}} \in \mathbb{R}^{N_x \times N_y \times N_z}$ can be expressed as:

$$\tilde{\mathbf{I}} = \tilde{\mathbf{I}}^k + \tilde{\mathbf{R}}. \quad (13)$$

As with OMP3D the aim is to minimise the norm of the residual at each step to ensure an optimal representation of $\tilde{\mathbf{I}}$ in $\mathbb{V}_k = \text{span}\{\mathbf{d}_{\ell_n^x}^x \otimes \mathbf{d}_{\ell_n^y}^y \otimes \mathbf{d}_{\ell_n^z}^z\}_{n=1}^k$ is achieved. This is accomplished by approximating $\tilde{\mathbf{R}}$ in $\mathbb{V}_k$ then subtracting the resulting component from $\tilde{\mathbf{R}}$ ensuring that $\widehat{\mathrm{P}}_{\mathbb{V}_k} \tilde{\mathbf{R}} = 0$.

To illiterate this, at each atom selection iteration the following steps are undertaken to project $\tilde{\mathbf{R}}$ on $\mathbb{V}_k$. Using the selected atoms $\{\mathbf{d}_{\ell_n^x}^x \otimes \mathbf{d}_{\ell_n^y}^y \otimes \mathbf{d}_{\ell_n^z}^z\}_{n=1}^k$, and setting $\mathbf{T}^0 = \mathbf{0}$, $\tilde{\mathbf{R}}^0 = \tilde{\mathbf{R}}$ and $j = 1$, select the tuple of indices such that:

$$\ell_j^x, \ell_j^y, \ell_j^z = \underset{\substack{n=1\dots k \\ s=1\dots k \\ t=1\dots k}}{\mathbf{argmax}} \left| \langle \mathbf{d}_{\ell_n^x}^x \otimes \mathbf{d}_{\ell_s^y}^y \otimes \mathbf{d}_{\ell_t^z}^z, \tilde{\mathbf{R}}^{j-1} \rangle_{3D} \right|, \quad (14)$$

and compute:

$$t(j) = \langle \mathbf{d}^{x}_{\ell^{x}_{j}} \otimes \mathbf{d}^{y}_{\ell^{y}_{j}} \otimes \mathbf{d}^{z}_{\ell^{z}_{j}}, \widetilde{\mathbf{R}}^{j-1} \rangle_{3D}, \tag{15}$$

$$\widetilde{\mathbf{R}}^{j} = \widetilde{\mathbf{R}}^{j-1} - t(j)\mathbf{d}^{x}_{\ell^{x}_{j}} \otimes \mathbf{d}^{y}_{\ell^{y}_{j}} \otimes \mathbf{d}^{z}_{\ell^{z}_{j}}, \tag{16}$$

$$\widetilde{\mathbf{T}}^{j} = \widetilde{\mathbf{T}}^{j-1} + t(j)\mathbf{d}^{x}_{\ell^{x}_{j}} \otimes \mathbf{d}^{y}_{\ell^{y}_{j}} \otimes \mathbf{d}^{z}_{\ell^{z}_{j}}. \tag{17}$$

Then set $j \leftarrow j + 1$ and continue the process until, for a predefined tolerance error ϵ, the condition $\left\| \widetilde{\mathbf{T}}^{j} - \widetilde{\mathbf{T}}^{j-1} \right\|_{3D} < \epsilon$ is satisfied. The asymptotic exponential convergence of this projection step is proved in (Rebollo-Neira & Sasmal, 2016).

4.1.3 Image Partitioning

It is not feasible from a computational perspective to approximate the whole image using general dictionaries. For this reason, the image is partitioned into small blocks of size $N_x \times N_y \times N_z$. For instance, a $256 \times 256 \times 3$ image, partitioned into blocks of size $8 \times 8 \times 3$ would yield 1024 blocks to be processed.

Given that after each blocks approximation there is a decomposition in the form of (12), the blocks simply need to be assembled to get the approximation for the whole image. Mathematically, given the independent decompositions in the form:

$$\tilde{\mathbf{I}}_{B}^{k_B} = \sum_{n=1}^{k_B} a(n)\mathbf{d}^{x}_{\ell^{x}_{n}} \otimes \mathbf{d}^{y}_{\ell^{y}_{n}} \otimes \mathbf{d}^{z}_{\ell^{z}_{n}},$$

where B represents the block being processed, the approximation of the whole image is assumed by

$$\tilde{\mathbf{I}}^{k} = \cup_{b=1}^{B} \tilde{\mathbf{I}}_{B}^{k_B}.$$

4.2 Objective Quality Measures

An important part of measuring the effectiveness of a sparse image approximation, let alone any signal compression, is measuring the quality of the resulting transformation. There are numerous Image Quality Assessment (IQA) methods widely used today, these can be split into two established categories: subjective and objective. Subjective quality assessment is a measurement of quality the user perceives and is obtained by a matter of opinion (Patil & Sheelvant, 2013). This is ideal since the main goal of image quality assessment is evaluating the adverse effects in quality as perceived by the user. The most common subjective quality assessment technique named the Mean Opinion Score (MOS) has been used for many years. It is a method in which a suitably sized group of observers are shown the images and are asked to rank image quality on a scale which is either continuous or discrete (Huynh-Thu, Garcia, Speranza, Corriveau, & Raake, 2011). However, MOS is inconvenient, time-consuming and impractical, so for measuring the quality of this application two extensively used objective measures will be (Wang, C. Bovik, & Lu, 2002).

4.2.1 Peak Signal to Noise Ratio (PSNR)

The Peak Signal to Noise Ratio and the common intermediate calculation Mean Squared Error (MSE) are the two "most widely used non-perceptual objective quality assessment metrics due to their direct application on images" (Al-Nuaim & Abukhodair, 2011). Although there is criticism that the measurement does not correlate well with user-perceived image quality (Wang et al., 2002, Yogita V & Y. Patil, 2015), other studies including the study from the Video Quality Expert Group which at the time represented "the largest single video quality study undertaken to date" concluded that the performance of the PSNR measurement is statistically indistinguishable from other, more sophisticated, IQA methods (VQEG, 2000). Despite the reasons against, due to its low compute overhead and the numerical nature of the PSNR, it is also a suitable input variable to calculating a tolerance to break the routine once a desired quality level has been reached.

From a mathematical perspective, the PSNR represents the ratio between the maximum possible power of a signal, which in this case is the bit depth of a pixel, and the power of distortion, which is calculated by the MSE between the original image and the approximation. Defining this mathematically for a 2D image we have:

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[\,I(i,j) - A(i,j)]^2 \tag{18}$$

Where *m, n* are the height and width of the Image *I* and Approximation *A* respectively.
This is then used to calculate the PSNR such that:

$$PSNR = 10\log_{10}\left(\frac{\left(2^b - 1\right)^2}{MSE}\right) \tag{19}$$

Where b is the number of bits required to represent the intensity of pixels in the image, e.g. for an 8-bit image the numerator would become 255^2.

As a concluding note on PSNR, for there to be some relationship to user perceived quality, we can use a table from J. Klaue's paper (J. Klaue, 2003) which maps a discrete 5-point MOS score to a PSNR in decibels. From this, let us assume that a PSNR of > 40 is of 'high quality' for our experiments.

PSNR [dB]	MOS
> 37	5 – Excellent
31-37	4 – Good
25-31	3 – Fair
20-25	2 – Poor
< 20	1 - Bad

Table 4-1 - A Possible PSNR to MOS conversion from (J. Klaue, 2003)

4.2.2 Structural Similarity (SSIM)

Another more sophisticated objective IQA is the Structural Similarity (SSIM) index described in "Image Quality Assessment: From Error Visibility to Structural Similarity" (Wang, Conrad Bovik, Rahim Sheikh, & P. Simoncelli, 2004). This method, unlike the PSNR attempts to predict the perceived quality of an image or video by considering the strengths of the Human Visual System (HVS) which is hypothesised to be well suited to extracting structural information (Wang, Conrad Bovik, Rahim Sheikh, & P. Simoncelli, 2004). The paper describes the structural information of an image as the objects in the scene independent of luminance and contrast. As such the algorithm has three distinct steps, each calculating a comparison with regards to luminance, contrast and structure.

For completeness, the derivation of the SSIM formula is defined (Wang, Conrad Bovik, Rahim Sheikh, & P. Simoncelli,, 2004).:

Firstly, the luminance comparison formula:

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1},$$

where μ_x has the conventional definition of mean ($\mu_x = \frac{1}{N}\sum_{i=1}^{N} x_i$) of the pixel intensity set x and C_1 is a stabilising constant typically,

$$C_1 = \left(K_1\left(2^b - 1\right)\right), \quad K_1 < 1.$$

For the contrast comparison, the mean of each signal calculated in the step for the luminance comparison is removed from the signal. The standard deviation of the resulting vector is deemed the estimate of the contrast and is used to normalise the vector for the structural comparison step:

$$c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}, \qquad \sigma_x^2 = \frac{1}{N-1}\sum_{i=0}^{N}(\boldsymbol{x} - x_i)^2,$$

where C_2 is of the same form as C_1.

The structural comparison is then defined by:

$$s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}, \qquad \sigma_{xy} = \frac{1}{N-1}\sum_{i=0}^{N}(x_i - \mu_x)\left(y_i - \mu_y\right),$$

where σ_{xy} is a measure of the correlation of structural information between the approximated image and the original image and $C_3 = \frac{C_2}{2}$. We can then define SSIM as a weighted function of the above comparison formulas. For simplicity assume all are given equal weighting of 1, then we get the special case formula:

$$SSIM(x,y) = \frac{\left(2\mu_x\mu_y + C_1\right)\left(2\sigma_{xy} + C\right)}{\left(\mu_x^2 + \mu_y^2 + C_1\right)\left(\sigma_x^2 + \sigma_y^2 + C_2\right)}.$$

The SSIM is bound by three conditions which is why it is desirable to use:

- It is symmetrical, so SSIM(x,y) = SSIM(y,x),
- Boundness since SSIM(x,y) ≤ 1,
- Unique Maximum since SSIM(x,y) = 1 if and only if x = y.

4.2.3 Sparsity Ratio

Another objective measurement for this application, albeit not an Image Quality Assessment measure, is the Sparsity Ratio which is an assessment of data reduction. This expresses how sparse the transformed domain is in comparison to the original range of pixels. This is simply defined by the equation:

$$SR = \frac{\text{Number of pixels in the original image}}{\text{Number of coefficients in the approximation}}. \quad (21)$$

Summarising the aforementioned objective measures, the aim of the application is to maximise the SR whilst achieving a 'high' PSNR and a SSIM Index value as close to 1 as possible.

4.3 Introduction to MATLAB and MEX

MATLAB, short for 'Matrix Laboratory', is a program that is widely used in the scientific and engineering community "to analyse and design the systems and products transforming our world." (Mathworks, 2017). The MATLAB language naturally enables the use of vectors and matrices to simplify computational mathematics. As already mentioned, because of the speed and ease of building prototypes of complex algorithms in MATLAB, the routines for the sparse image approximation have already been built and tested in MATLAB.

Since MATLAB is optimised for operations on vectorised variables, routines that consist of nested loops are very slow. This is why it is encouraged to use vectorisation instead of loops wherever possible. Vectorised code looks more like natural mathematical notation and therefore has a better appearance. It is less error-prone since it requires fewer lines of code, and has better performance since MATLAB treats variables as arrays (Mathworks, 2017). Although the routines considered in this report are vectorised where possible, due to the greedy nature of the methods the nested loops are inevitable.

```
a = rand(1, 100)
b = rand(1, 100)
c = a + b
```

Figure 4-1 - Example MATLAB Program adding two vectors

```
for (int i = 0; i < 100; i++){
      c[i] = a[i] + b[i];
}
```

Figure 4-2 - Example addition of two arrays in Java, using a for loop

Figure 4-1 demonstrates an example of element-wise array addition using vectorised notation, whereas Figure 4-2 illustrates how other programming languages would tackle the problem using a for loop.

MATLAB also allows programmers to use C, C++ or Fortran and call this code from within MATLAB as if they were normal functions. This is done using MATLAB Executable files, also known as 'MEX' files. MEX files are compiled using the inbuilt 'mex' command along with any arguments and required libraries. In order for MEX files to compile they must satisfy two conditions:

- they must include the header file, '`mex.h`' which declares the functions defined in MATLABs Application Programming Interface (API).
- they must have a gateway function '`mexFunction`' which is the entry point into the program.

The '`mexFunction`' entry point takes four arguments which specify the inputs and outputs of the program. There are two integer arguments: nlhs, which we assign to declare the number of variables we are returning to MATLAB via plhs; and nrhs, which is set on entry and specifies the number of variables which were passed to the function in MATLAB. Both plhs and prhs are arrays of mxArray pointers which store the output variables and input variables respectively. Note that prhs is declared const since MATLAB does not allow us to modify the input variables. Figure 4-3 is an example of a simple MEX file:

```
    void testFunction(double* input, double* output){/*do something*/}

    void mexFunction(int nlhs, mxArray *plhs[],
            int nrhs, const mxArray *prhs[]) {
            // Condition the input variables so that they are in
            // the correct format for the C/C++ program
            double* input = mxGetPr(prhs[0]);
            int height, width = 1;
            plhs[0] = mxCreateDoubleArray(height, width, mxREAL);
            double* output = mxGetPr(plhs[0]);

            // Run the C/C++ function
            testFunction(input, output);
    }
```

Figure 4-3 - Example structure of a MEX file, with the mexFunction entry point

4.4 Introduction to C++

Although C++ is new to the author, the semantics of C++ are very similar to other programming languages such as Java, so only the key differences found whilst becoming familiar are discussed. C++ was built on top of the procedural programming language C and provides object-oriented programming, exception handling, the C++ standard library, amongst much more. The C++ standard library provides collections of functions that encapsulate common patterns for convenience; these include but are not limited to facilities for I/O, vector containers which are the equivalent of arraylists and tools for working with threads.

The most distinct difference when using C++ is the necessity to manage memory manually. Unlike Java, there is no automatic garbage collection. Thus, allocation and deallocation of memory are the responsibility of the developer. This means issues such as memory leaks, segmentation faults and other memory-related errors are far more likely while developing in C++. However, this is the cost of using C++ and its ability of fine-grained control over the memory it uses. The language uses pointers, which are, simply put, objects storing an address of a location in memory. When using pointers, memory is accessed directly. Hence it is the quickest way to access a piece of data.

Another variation is the way variables can be passed to methods. Java is always pass-by-value, but this value could be a reference to an object. Whereas C++ supports pass-by-value and pass-by-reference, this allows arguments that are given to the function to be modified or completely redefined. It also allows an approach where the output of the method is assigned to an input variable, instead of the function returning anything.

4.5 Introduction Graphical Processing Units and CUDA

At time of writing there are two prominent general purpose GPU (GPGPU) programming frameworks that are widely used and have made significant progress in recent years towards General Purpose GPU programming (Khronos Group, 2017) OpenCL which is developed by the Khronos Group and CUDA developed by Nvidia. OpenCL has the benefit of being able to work across a wide range of platforms and compute devices from many different vendors, making the application very portable. However due to the introduction materials, hardware available and comprehensive online resources available for Nvidia's CUDA framework, the decision was made to use this for the application. For clarity, this does mean it will only work on computers with Nvidia hardware but it does have the benefit of being able to use the latest features of the GPU.

Developing applications that use the GPU are fundamentally different to programming applications for the CPU. Rather than the programmer having to think linearly when developing code, the application must be structured in a way that maximises parallelism. A CPU core can only execute one instruction stream at a time, but very quickly, so that it gives the appearance of multiple threads being executed at once; however GPUs are designed to allow multiple threads to run in parallel on the same dataset.

A typical GPU is composed of a number of Streaming Multiprocessors (SMs) which are responsible for scheduling and execution of groups of 32 threads called warps (NVidia Corporation, 2017). These threads start at the same address in the program, but each have their own program counter and register state which allows them to take independent paths in the program execution. In parallel programming this is often known as thread divergence. GPUs follow a Single Instruction, Multiple Thread (SIMT) architecture, so each thread in the warp will execute the same instruction at the same time. If the threads do diverge and take a different path, then the threads taking an alternative path are disabled until the active threads have completed the execution path and can converge.

4.5.1 Parallel Random Access Machines (PRAM)

The PRAM model of parallel computing can be used to effectively design good parallel programs that optimise memory read and writes and helps to reason the complexity of algorithms. The PRAM model builds on Random Access Machine (RAM). RAMs are convenient since it is easy to quantify the time complexity and spatial complexity. Time complexity is the number of instructions executed on the machine and spatial complexity is the number of cells used (Tvrdik, 1999). Moreover, PRAM can be defined as a system (M, S) where M is an unbounded number of RAMs each with its own local memory and capable of knowing its own index M_i and S represents the shared memory cells which can be accessed on any of the machines. Each machine in the PRAM instruction executes a 3-phase cycle which involves reading memory, local computation and writing to memory if needed. PRAM models can be categorised as one of three groups depending on the read and write behaviour of the machine:

- Exclusive Read / Exclusive Write (EREW) is a machine in which no two processors can access or write to the same piece of memory simultaneously;
- Concurrent Read / Exclusive Write (CREW) is a machine which allows all processors to read the same memory cell concurrently, however prevents writes to the same cell;
- Concurrent Read / Concurrent Write (CRCW) is a machine which allows all processors to read and write to the same memory cell in parallel.

Concurrent read is a well-defined operation as shown in Section **4.5.3.1**, however Concurrent Write has various mechanisms. If all the machines are writing the same value, then all processors can write with no consequence, known as common CRCW. Another approach is assigning priorities to each processor and the one with the highest priority is allowed to write. Finally, an arbitrary processor is chosen and allowed to complete the write, however this can lead to undefined behaviour.

For the sake of an example, a CREW PRAM model would be suitable for defining the matrix multiplication algorithm. The concurrent read allows the processors to access any element of the component matrices. They are modelled to exclusively write, since each processor would write to one specific element in the resulting matrix. The running total of each element would be stored in the processors local memory.

Parallel complexity of the PRAM can then be determined for the algorithm based on the number of RAMs used in the model. Whilst spatial complexity is calculated based on the number of shared cells in use. As now discussed we can realise this model in CUDA since each thread has its own index and local memory for performing calculations.

4.5.2 Compute Unified Device Architecture (CUDA)

CUDA is a parallel computing architecture and application programming interface invented and developed by NVidia, which enables programs to use the capabilities of the GPU to significantly increase performance when applied to certain applications. Most of the information in this section has been obtained from the NVidia C Programming Guide (NVidia Corporation, 2017) unless otherwise stated. CUDA has a three-level hierarchy, the largest unit being a grid, which can be at most a 3-dimensional container of thread blocks. Thread blocks can also be at most a 3D container of threads with individual

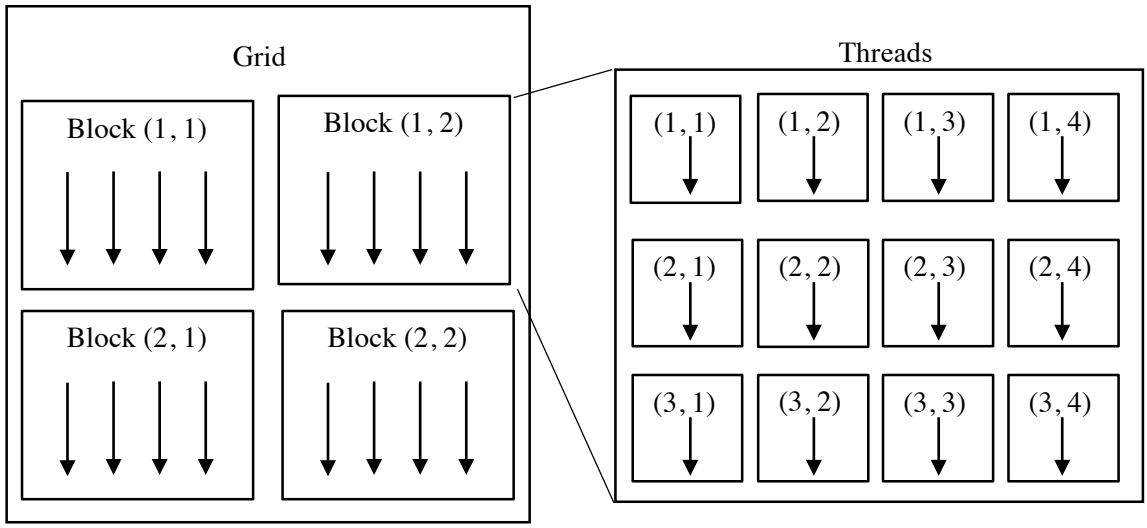


Figure 4-4 – Representation of Grids, Blocks and Threads in the CUDA Hierarchy

threads being the smallest unit. There are other limitations on the maximum size of grids and blocks but these vary between devices. Figure 4-4 briefly demonstrates the hierarchy for grid size of 2x2 and a block size of 3x4.

CUDA has an API that works synergistically with C/C++, and as such it helps programmers to become familiar with the language quicker since there is no change in the general syntax. To compile CUDA code, the NVCC compiler must be used which translates all the device code and CUDA syntax into C++ source code before passing it to the native compiler – on Linux, this is usually g++. As a side note, it is common CUDA terminology to call the code that is executed on the GPU 'device' code and the code that is executed on the CPU the 'host' code, this terminology will be used from now on.

Functions that initiate code to be run on the device are called kernels and are decorated by the '`__global__`' declaration specifier. Kernels can be executed either from host or device code hence the global naming. When a kernel is executed it must be configured, which specifies the number of blocks and number of threads per block that should be launched. CUDA has special syntax (denoted <<<… , …>>>) to provide this information which simplifies the launch process instead of calling multiple APIs, this is automatically translated by the NVCC compiler. One important note is the parameters that are passed to a kernel must be pointers to a memory address on the GPU, they cannot be pointers on host memory. Figure 4-5 demonstrates an example of a kernel launch whilst configuring the number of blocks to launch in a grid, and the number of threads each of those blocks should contain.

```
int main(){
      int numberOfBlocks //Can either be int or dim3
      dim3 threadsPerBlock(8,8,3); //Dimensions of the block
      kernelToBeCalled<<<numberOfBlocks, threadsPerBlock>>>(params, …);
      return 0;
}
```

Figure 4-5 - Code example of a CUDA Kernel launch with the kernel configuration being passed between <<<… , …>>>

There are two other declaration specifiers CUDA uses. For methods that can only be executed on the host there is the '`__host__`' specifier. Then there is the '`__device__`' specifier for functions that can only be called from the device. If a function has no specifier it is implicitly declared as a host only method. It is also possible for a function to have both the '`__host__`' and '`__device__`' specifier if it works on both devices, the NVCC compiler will automatically create code for both the CPU and GPU in this scenario.

To be able to maximise the parallelism there must be a mechanism so that each thread can access a unique area of memory to use for performing operations. As with the PRAM model where each machine must know its index, CUDA provides two variables accessible from device code named '`blockIdx`' and '`threadIdx`' which will have `x`, `y`, and `z` components dependent on the dimensions configured when calling the kernel. Each thread will have a unique coordinate within the block which can be used as index to access areas in memory unique to that thread, this is also demonstrated in Figure 4-6. Note that the same idea is transferable for the relationship between blocks and grids.

Matrix multiplication is a task that can be easily parallelized since each element can be calculated independent of each other, so given sufficiently sized matrices, performing these operations on the GPU should generally be quicker than performing them on the CPU. The below code snippet is an example of using the '`threadIdx`' variable to access the elements needed to multiply two matrices and output the result to matrix '`C`'. Assume for this example the block size is the same size of the matrix's being multiplied. If this were to be done on the CPU then three loops would need to be required; iterating between the rows, columns, and when summing the products for each element.

```
__device__ void multiply(Matrix* a, Matrix* b, Matrix* c){

    int x = threadIdx.x; //row
    int y = threadIdx.y; //col

    if (y < a->width && x < b->height){
        double cellSum = 0;
        for(int i = 0; i < a->height; i++){
            cellSum += a->elements[a->height * i + y] *
                                b->elements[b->height * x + i];
        }
}
```

Figure 4-6 - Example of a Matrix Multiplication function that uses the threadId variable for each thread to access its individual area of memory

4.5.3 Memory Access

There are multiple kinds of memory available to a GPU, however, here I will only focus on global memory, local memory and shared memory. Global memory is accessible by any of the threads in any streaming multiprocessor, but the latency in comparison to shared memory is roughly 100 times slower since shared memory is on chip (Harris, Using Shared Memory in CUDA C/C++, 2013). Despite its name suggesting otherwise, local memory resides in a reserved area on the GPUs global memory space, so the latency is the same as global memory.

In terms of size, typical shared memory is 48kb, however, since CUDA compute capability (CC) 3.x this is configurable so that this can be changed to 32kb or 16kb, increasing L1 cache size to 32kb and 48kb respectively. Local memory has been 512kb since CC 2.0, whereas global memory depends on the individual device, but is much larger, typically in the Gigabytes.

Since shared memory is so much quicker it is desirable to use it as much as possible to reduce the overhead of reading and writing to global memory. Shared memory is structured as a series of banks, since CUDA 2.x there are 32 banks for each block of shared memory. These banks allow concurrent access to data when there is only one address being read per bank.

In CUDA versions later than 3.x there is an option to configure the bank sizes so that they are either four or eight bytes in size. Eight bytes is more favourable when dealing with double precision numbers since the number isn't split over two banks, hence reads from two different banks are not required. Figure 4-7 is an example of how addresses are split into the banks assuming a bank size of eight bytes.

```
banks              0                    1                  …              31
address   0 1 2 3 4 5 6 7 | 8 9 10 11 12 13 14 15 |   …     | 248 249 … 254 255
address 256 257 … 262 263 | 264 264  …   270  271 |   …     | 504 505 … 510 511
address 512 513 … 518 519 | 520 521  …   526  527 |   …     | 760 761 … 766 767
```

Figure 4-7 - Example of how addresses are split between memory banks with a bank size of 8 bytes

It is good practice to coalesce memory so that the number of read transactions from global memory are kept to a minimum. This is opposed to stridden access where it is not possible for the hardware to combine accesses. For instance, consider a routine where each thread is responsible for summing up a matrix like shown in Figure 4-8, where the number represents the address in memory and the arrow represents the order the routine is adding the elements. It is more efficient to make use of sequential reads in memory since the hardware is capable of grouping accesses together. Figure 4-9 is a code example of how a thread might sum the matrix that is coalesced. For completeness figures 4-10 and 4-11 show how a thread summing a matrix using strides is used. Results from (Harris, 2013) show that coalescing memory increases performance time by 2.5 compared to using strides.

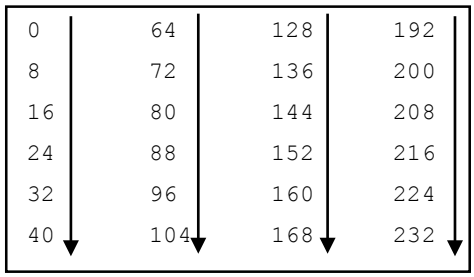


Figure 4-8 - Example matrix using coalescing

```
__device__ void sumMatrix(){
  double sum = 0;
  for(int j = 0; j<this.width;j++){
   for(int i = 0; i<this.height;i++){
       elements[j * height + i];
   }
  }  return sum; }
```

Figure 4-9 - Code example of summing a coalesced matrix

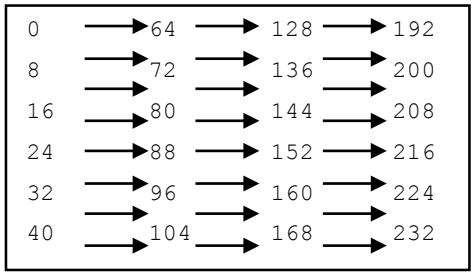


Figure 4-10 - Example matrix by striding

```
__device__ void sumMatrix(){
  double sum = 0;
  for(int i = 0; i<this.height;i++){
   for(int j = 0; j<this.width;j++){
       elements[j * height + i];
   }
  }  return sum;}
```

Figure 4-11 - Code example of summing a matrix by striding

4.5.3.1 Shared Memory Bank Conflicts

If more than one thread attempts to read different addresses from the same bank, then this is considered a bank conflict and the hardware is forced to serialise the requests into conflict free requests. For this reason it is desirable to reduce the number of bank conflicts when reading from shared memory.

Figure 4-12 demonstrates an ideal pattern where each thread linearly accesses memory unique to that thread. This can easily be done by using the threadId as a multiplier for the index of an array where the bank is the same size as the data type of the array. Figure 4-13 demonstrates an undesirable 2-way bank conflict where each thread is trying to access a 4-byte piece of data stored in an 8-byte sized bank. This results in each bank having to serve two requests consecutively instead of linearly as in figure 4-12.

This would normally cause problems if all the threads in a block wanted to access the same piece of data, however devices higher than CUDA 2.x have banks with broadcast and multicast capabilities which mean accesses from multiple threads to the same addresses in one bank can be served simultaneously reducing the number of conflict-free requests needed. That is to say, if all 32 threads access the same area in memory at the same time, the responsible bank would broadcast the data at the address to all threads at the same time. An example of this is shown in figure 4-14.

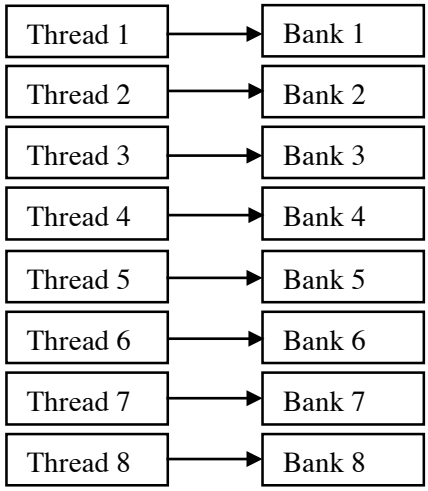


Figure 4-12 - Accessing memory banks with no conflicts

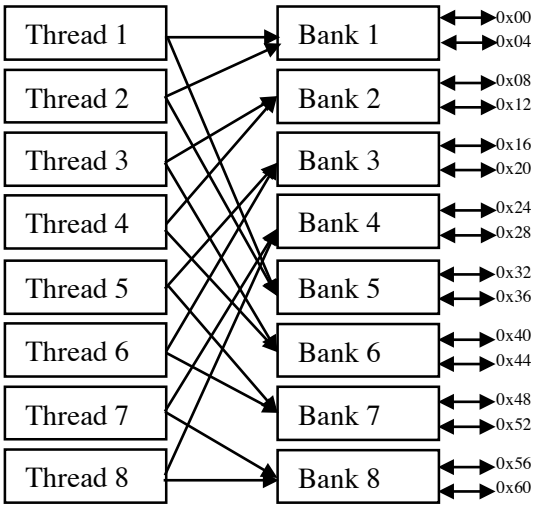


Figure 4-13 - Threads accessing banks causing a 2-way bank conflict

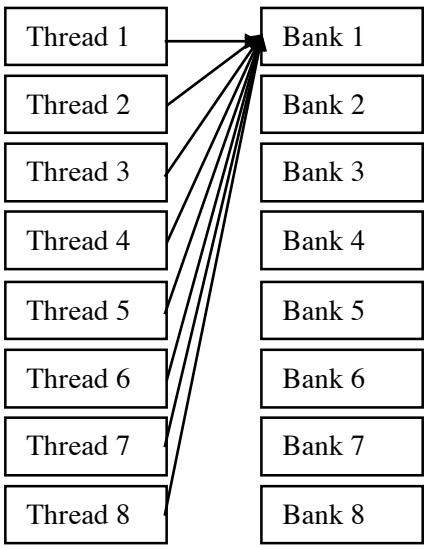


Figure 4-14 - Bank broadcasting the address since it is requested from all threads

4.5.4 Block Level Synchronisation

Warps by design are synchronised on an instruction level (known as instruction level parallelism), but physically it is impossible for threads to complete at exactly the same time, meaning there will be some variation in the time it takes for warps to complete. If a block is not exactly the size of one warp (32 threads), it is required for all the threads in the block to be synchronised at certain stages in the program to prevent warps operating on data in the wrong order. To prevent undefined behaviour of warps operating on data that has not been completely processed by another warp, it is often required for a block to be synchronised at certain points. The '`__syncthreads()`' is a block level synchronisation barrier in which threads are not allowed to proceed past the barrier until all threads in the block have reached the same statement.

Care must be taken to prevent deadlock situations such as scenarios where the warp is waiting for all threads to reach a certain point, but due to the divergent paths taken by the threads the program is unable to continue. The code in figure 4-15 demonstrates a deadlock scenario which would cause the program to hang since it is not accessible to all of the threads.

```
if(threadIdx.x % 2 == 0){
      …     …     …     …
      __syncthreads();
}
```

Figure 4-15 - Code example of deadlock situation

5 Deliverable

5.1 Approach

The deliverable is essentially delivered in two parts. The first part is the collection of MEX C/C++ files that aim to improve the performance of the routine from within MATLAB. The second is the executable command line application that utilises the GPU for acceleration.

The approach taken with the MEX files, with the intention of becoming more accustomed to the language, was to start with the most computationally intensive function and build outwards. Essentially, building prototype functions and taking an evolutionary approach until the whole routine was covered. This is beneficial since it is possible to call each of these functions individually from MATLAB i.e. the subroutines are reusable. The GPU application took a similar approach, however since the whole routine is compiled it is not possible to immediately reuse functions without code change.

As a foreword on common attributes in this section, all coefficients and matrices will be stored in double precision. The aim is to achieve a high Sparsity Ratio, and this can only be achieved when there is minimal loss of error in the routine. Also, the matrices will be stored in column major order, the same as MATLAB and Fortran, although this is typically unconventional for C++ programs.

5.2 MEX-Files

The first routine to be realised in C++ was the SPMP routine, termed here SPMP3D. As said, the most intensive parts of the routine were created first. Not only because these parts were small but because they could benefit from performance speed up significantly. Mainly because they included nested loops, which as mentioned in Section **4.3**, fails to perform well in MATLAB. To ascertain the bottlenecks in the program, MATLABs profiling capabilities were used which showed the amount of time the CPU spent on each function, and how many calls to that function were made.

Profiling the MATLAB routine gave the results in Figure 5-1. The self-time field is the most import since it shows where the CPU spend the most time, total time includes calls to other routines. From the result, it is clear IP3D and IPSP3D are the most time-consuming functions. This is expected since these functions are the realisation of formula (1) from Section **3.2**. It is clear from observation that summing three dimensions of arrays is a heavy operation, and yet these functions are called 256,272 and 888,546 times respectively.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
test_SPMP3D_Dan	1	374.906 s	2.995 s	
SPMP3D	34560	370.196 s	31.108 s	
ProjMP3D	256272	160.995 s	16.598 s	
IP3D	256272	153.792 s	153.792 s	
IPSP3D	888546	123.376 s	123.376 s	

Figure 5-1 - Profiling results of a 1730 x 1280 colour image with block size of 8 in MATLAB

5.2.1 Inner Product, Self-Projected (IPSP3D)

For simplicity, I chose to work on IPSP3D first, since it deals with vector-matrix-vector multiplication opposed to matrix-matrix-matrix multiplication. This function is used when calculating the inner product as shown in formula (14) in Section **4.1.2**. The effect on performance of writing this one function in C++ is astounding, the individual function performed 21x quicker than the MATLAB equivalent as shown in figure 5-2. From a routine level, this decreased the routine time by 40%.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
test_SPMP3D_Dan	1	226.999 s	1.966 s	
SPMP3D	34560	223.328 s	26.471 s	
IP3D	256272	139.501 s	139.501 s	
IPSP3D_mex (MEX-file)	888541	5.811 s	5.811 s	

Figure 5-2 - Profiling results of a 1730 x 1280 colour image using a C++ implementation of IPSP3D

5.2.2 Inner Product 3D (IP3D)

As with the IPSP3D function the next function to be realised was the IP3D, which is used for formula (6) in the routine, and determines the indices of the best atom for this iteration. Unfortunately, as shown in Figure 5-3, the raw adaptation of this function in C++ was actually slower, by 33%. This can almost certainly be pinned down to the fact MATLAB excels with vectorised operations and this function calculates matrix-matrix-matrix multiplication. It was clear at this point in order to become more efficient than MATLAB external libraries are needed.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
test_SPMP3D_Dan	1	293.938 s	2.168 s	
SPMP3D	34560	290.221 s	27.785 s	
IP3D_mex (MEX-file)	256273	204.159 s	204.159 s	

Figure 5-3 - Profiling results of a 1730 x 1280 colour image using a C++ implementation of IP3D with no optimization

5.2.3 BLAS Library Utilisation

To tackle the issue of IP3D taking longer I used the basic linear algebra subprograms (BLAS) library. The original routines were developed for Fortran (Lawson, Hanson, Kincaid, & Krough, 1979), and offers highly optimised implementations of vector and matrix operations. The library has three levels of operations; level 1 performs scalar and vector-vector operations, level 2 performs matrix-vector operations, and level 3 performs matrix-matrix operations. The library provides implementations for single precision and double precision, but only double precision will be used since accuracy is quintessential for effective sparse image approximation.

BLAS is often very difficult to set up on computers since it is fine tuned for each system, fortunately however MATLAB is shipped with a BLAS library that can simply be included when compiling with the MEX compiler.

In fact, on consideration the only parts of the routine I felt could be greatly improved by using BLAS are the matrix-matrix multiplications, these do happen regularly. The matrix-matrix BLAS function used is called dgemm, which is a level 3 operation, standing for 'Double precision general matrix-matrix multiplication' : (Dongarra, Duff, Croz, & Hammarling, 1989). More specifically, the dgemm function performs the following operation:

$$C = \alpha \, op(A) \, op(B) + \beta C$$

```
dgemm(op1,op2, m, n, k, alpha, a, LDA, b, LDB, beta, c, LDC);
```

Figure 5-4 - Example function call to the BLAS level 3 matrix-matrix multiplication function 'dgemm'

The function call, shown in Figure 5-4 takes 13 parameters which are all pointers to addresses in memory. At a glance, the purpose of each of the parameters are indicated here:

- op1 and op2 are chars which indicate the operation applied to the matrices in the formula. This can either be 'N' for no operation, 'T' to transpose the matrix or 'C' to take the complex conjugate transpose.
- m, n and k represent the number of rows of op(A), number of columns of op(B), and the number of columns of op(A) and rows of op(B). This determines the size of the resulting matrix.
- alpha is a scalar which multiplies the result of op(A) op(B).
- beta is some scalar which multiples any existing elements of C. By default, this is zero, but if it is 1, then the result of alpha × op(A)× op(B) is added onto the current values of c.
- a and b are the matrices to multiply whilst c is the output matrix. Since it is double precision, these are stored as double arrays.
- LDA, LDB and LDC represents the leading dimensions of the matrices, normally the height.

When implementing IP3D with a BLAS routine the execution time was significantly decreased. As shown in Figure 5-5, there is roughly a 64% decrease in time when using these functions with BLAS compared to the results from figure 5-1. Interestingly, IPSP3D actually takes slightly longer when using BLAS, but this may be because of the cost object allocation and initialization used in this function.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
test_SPMP3D_Dan	1	132.997 s	2.141 s	
SPMP3D	34560	129.020 s	25.855 s	
IP3D_mex (MEX-file)	256272	43.609 s	43.609 s	
hnew3D	1063183	23.226 s	23.226 s	
IPSP3D_mex (MEX-file)	888559	8.481 s	8.481 s	

Figure 5-5 - Profiling results of a 1730 x 1280 colour image using a C++ implementation of IP3D with BLAS

5.2.4 Approximation for the Iteration (hnew3D)

Continuing this pattern of optimising the most computationally intensive routines, I moved on to hnew3D. This function is responsible for calculating the approximation for that iteration when using the chosen atoms, this is equation (15) from Section **4.1.2**. As with the previous two functions, due to the vector-vector multiplication in this function, it is a strong candidate to being optimised with BLAS. Implementing the hnew3d function in C++ with BLAS resulting in an 83% increase in function-level performance, this also shed ~25 seconds on the entire routine when using a block size of 8, as shown in Figure 5-6

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
test_SPMP3D_Dan	1	108.879 s	2.089 s	
SPMP3D	34560	105.111 s	25.526 s	
IP3D_mex (MEX-file)	256271	43.419 s	43.419 s	
IPSP3D_mex (MEX-file)	888548	6.262 s	6.262 s	
hnew3D_mex (MEX-file)	1063160	3.795 s	3.795 s	

Figure 5-6 - Profiling results of a 1730 x 1280 colour image using a C++ implementation of hnew3d with BLAS

At the current stage the structure of the C++ files are as shown in Figure 5-7, the figure introduces a commonOps file which aids with reusing functions that are used between all routines, this file alone is the interface for the BLAS routines. Some trivial accessor and mutator functions have been omitted for clarity. 'mex.h' and 'blas.h' are external header files included when compiling.

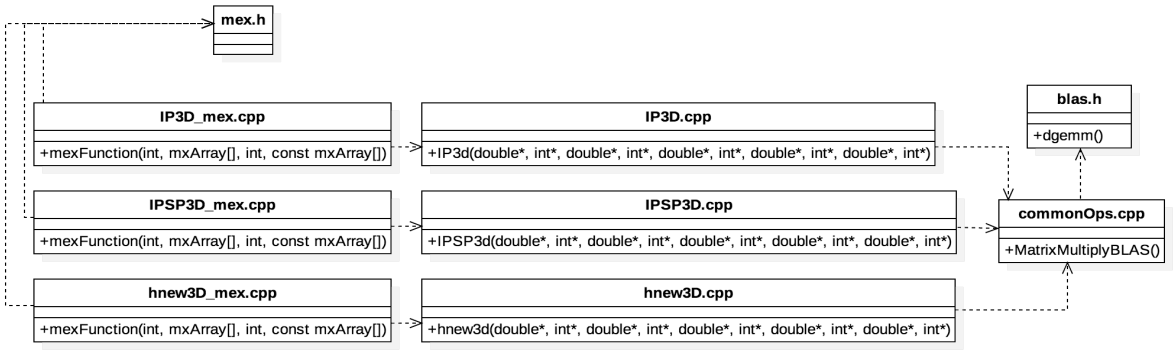


Figure 5-7 - UML diagram showing the dependencies between the current C++ files

5.2.5 Projection Stage (ProjMP3D)

The next function that had the largest self-compute time was the ProjMP function which is the realisation of Section **4.1.2** of this report and is responsible for ensuring the norm of the residual at each iteration is a minimum by projecting the residual on the space spanned by the chosen atoms. This is accomplished by using IPSP3D which has already been built, choosing the maximum atom and then calculating the approximation of using this atom by using hnew3D, which has also already been developed. Once we have this temporary approximation for *this* iteration, the approximation is subtracted from the residual and added to the *entire* projections approximation. This continues until the norm of the

residual has met some predefined criteria. This may be clearer in the outline code shown in Figure 5-8, some declarations have been omitted for clarity.

```
ProjMP3d(){

   for(int it = 0; it < max; it++){
      IPSP3d(re, reDim, v1, v1Dim, v2, v2Dim, v3, v3Dim, cc, ccDim);

      int n1 = max(cc,ccDim);
      double maxValue = std::abs(cc[0]);

      if (maxValue < tol2){ break; }

      cc = hnew3d( <varargs> );
      c[n1] += cc[n1];

      double nornu = 0;
      for(int k = 0; k < reDim[0] * reDim[1] * reDim[2]; k++){
        h[k]  += hnew[k];
         re[k] -= hnew[k];
        nornu += hnew[k] * hnew[k];
      }
      nornu *= delta;
      if (nornu <= toln){ break; }
   }
}
```

Figure 5-8 - Code outline of the ProjMP3D function

After profiling the results when using the ProjMP3D mex file, we can see a 82% decrease in execution time in Figure 5-9. So far when writing the MEX functions there has been a 60-80% decrease in execution time.

ProjMP3D	256271	24.361 s	15.076 s	
ProjMP3D_mex (MEX-file)	256273	4.181 s	4.181 s	

Figure 5-9 - Profiling results of ProjMP3D in MATLAB (top) and C++ (bottom)

Revisiting the updated class diagram in Figure 5-10, we now see 4 entry points from MATLAB; `IP3D_mex`, `IPSP3D_mex`, `hnew3D_mex` and `ProjMP3D_mex`. These all depend on the `mex.h` for the implementation of the entry point `mexFunction()`. `ProjMP3D.cpp` also includes the previously developed `hnew3D.cpp` and `IPSP3D.cpp`, however they bypass the mex entry point since

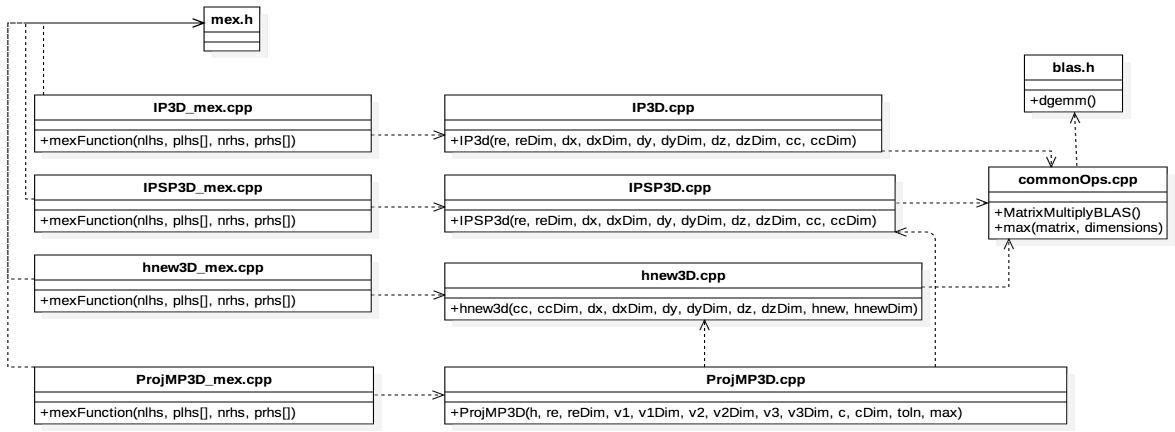


Figure 5-10 - UML diagram showing the dependencies between the current C++ files - ProjMP3D.cpp

no conversion from MATLABs `mxArray` object is required. The implementation makes further use of the `commonOps.cpp` file, via either hnew3D or IPSP3D files for reusability of common functions.

It is worth mentioning at this point, since both IPSP3D and hnew3D include the commonOps file. There would normally be a compiler error stating the functions in commonOps have been defined more that once. To prevent this the use of include guards has been utilised. These are compiler level macros that tell the compiler if something hasn't been defined, include it or else leave it out. In this case the definition is commonOps, as shown in Figure 5-11.

This completes all the subroutines required for the SPMP3D routine, so the next largest function that had the largest MATLAB self-execution time was the SPMP3D routine itself.

```
#ifndef COMMON_OPS
#define COMMON_OPS

 /* Common Ops Operations Here */

#endif
```

Figure 5-11 - Example of an include guard preventing compilation errors when including files multiple times

5.2.6 Self-Projected Matching Pursuit (SPMP3D)

Since SPMP3D is the highest level routine, the decision was made to include both the mexFunction() entry point and the actual SPMP3D function in the same file. This was not possible with the previous functions because they are included in other routines, thus the compiler would throw errors because there are multiple mexFunction definitions. SPMP3D is quite a large routine, but to summarise it:

- Calculates the inner product as shown using the formula (6) from Section **3.3**, and determine the best atom;
- Store the indices of the selected atom if they haven't already;
- Calculate the approximation for this iteration from using the selected atoms;
- Attempt to minimise the residual by projecting the approximation of the set spanned by the chosen atoms;
- Repeat until the error between the approximation and the image has reached some tolerance level.

A lot of the mentioned responsibilities are performed by some of the functions mentioned earlier in this section. So SPMP3D is responsible for resource allocation for the lifetime of the routine and calling these other functions with the correct arguments. It is also responsible for the iterating and determining when the routine should break;- most often when the norm of the residual is sufficiently small, however can also be because the maximum number of iterations has been reached.

Notably, the way the SPMP3D accepts its parameters is identical to the way the MATLAB function accepts its parameters, so the user may simply replace the function name without the need of adapting the arguments. A full dependency diagram for the SPMP3D routine showing all functions is shown in Appendix 1 (Section **10.1.1**).

5.2.7 Orthogonal Matching Pursuit (OMP3D)

At this point, the decision was made to also build the OMP3D algorithm. In a similar fashion to the SPMP3D algorithm the approach was start with individual functions and evolve the implementation. OMP3D can use a lot of the functions that were created because of SPMP3D, however there are some new functions that need to be created. Profiling the routine, the functions in Figure 6-12 are highlighted for complexity.

Reorthogonalize	57359	2.373 s	2.373 s	
Orthogonalize	57359	1.920 s	1.920 s	
kron	127006	1.788 s	1.788 s	
Biorthogonalize	57359	0.817 s	0.817 s	

Figure 6-11 - Profiling results showing standard timing for Reorthogonalize, Orthogonalize, Kron & Biorthogonalize

Starting by building the routine for Kronecker product and comparing the result in figure 6-13, we can see that there has been a four-fold increase in execution time. Although the performance of these functions are not a significant part of the routine, they still need to be implemented in C++,

kronecker (MEX-file)	127006	0.614 s	0.614 s	

Figure 6-12 - Profiling results showing the execution time for kronecker, the MEX implementation of kron

Comparing the various orthogonalization functions, there is a three-fold increase in performance for these functions as shown in Figure 6-14.

o_reorthogonalize (MEX-file)	57359	0.670 s	0.670 s	
kronecker (MEX-file)	114718	0.466 s	0.466 s	
biorthogonalize (MEX-file)	57359	0.303 s	0.303 s	

Figure 6-13 - Profiling results showing the execution time of orthogonalize, reorthogonalize and biorthogonalize

These functions were built in a similar format to those used in SPMP3D, that is each have their own MEX file containing the mexFunction entry point, however the actual function is kept in the commonOps file. For example, the file 'kronecker.cpp' only contains the mexFunction call that acts as the interface to the actual function call which is stored in commonOps. This is useful since it allows the functions to be reused within other routines in MATLAB. Figure 6-15 briefly describes the dependencies between the files for OMP3D, a full illustration can be found in Appendix 1 (Section **10.1.2**).

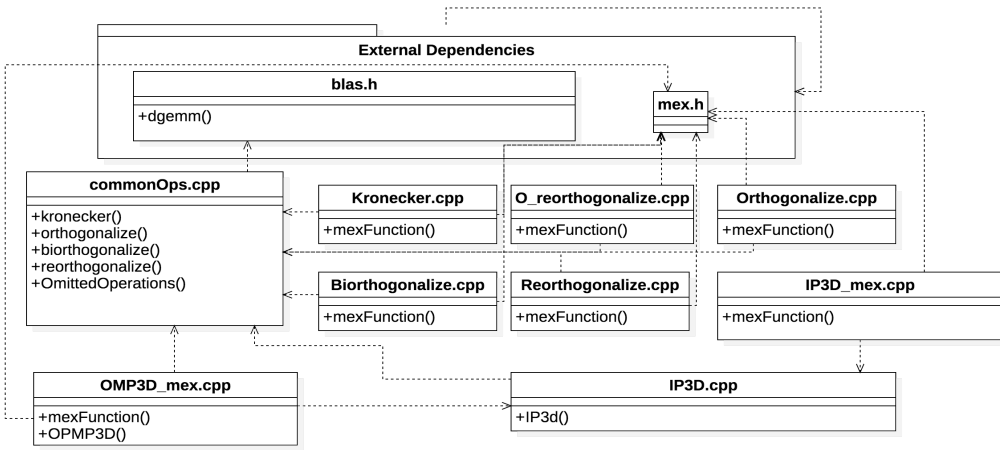


Figure 6-14 - Dependency diagram showing files involved in OMP3D routine

Deliverable | MEX-Files

5.2.8 Test Method

For testing, comparisons of the functions mentioned in the previous sections were compared to the results of their MATLAB counterparts. Essentially this is black box testing, as they were called from MATLAB and the inner workings of the functions irrelevant. Since the functions are designed to work with MATLAB, there is an essence of integration testing i.e. ensuring that the MATLAB calls to the C++ executables are successful. For testing during development, the C++ routines were used for approximation of small and very large images, on varying block sizes, however these are not recorded in this report. The **evaluation** section contains results comparing the performance of the MATLAB routines and MEX routines, where the quality of the approximations are identical. Thus, from a black-box testing perspective, these routines work as designed.

5.3 CUDA Implementation

The concept of the CUDA implementation is to use the GPU to be able to process multiple blocks in parallel by the streaming multiprocessors to each schedule and execute one block at a time. The hardware that has been used to develop and test this implementation is a NVidia GTX Titan, with 6GB of GDDR5 memory, 2688 cores over 14 SMs whereas the CPU is a 3.4GHz 4 core Intel i7-3770. Two distinct ways were considered for the program structure, considering the 48kb shared memory restriction per SM discussed in Section **4.5.3**. Both considerations only take into account a block size of 8 since it would not be possible to store dictionaries on the GPU shared memory with the larger block sizes.

1. The program could utilise more shared memory per block by using the memory for intermediate calculations in the block, minimising memory latency throughout the routine,
2. or seeing as each SM on this GPU had 192 CUDA cores, the program could attempt to concurrently run 3 blocks per SM at the cost of only storing the most essential variables in shared memory.

With option two, the drawback would be that some calculations would have to be computed using global memory as the storage mechanism between threads in a block. Since the number of required iterations for some blocks is undetermined and potentially large, for instance with images where it is not possible to achieve a high sparsity ratio. It's conceivable that the high latency of using global memory is amplified and is detrimental to performance. On the other hand, the routine would be able to run three concurrent blocks per SM, on this hardware that would be in total 42 blocks being processed simultaneously. Another point to be made is this is somewhat hardware specific. The GPU being used is CUDA compute capability 3.5, thus it has 192 CUDA cores per GPU (NVidia Corporation, 2017) and since each block has 64 elements the SM can effectively handle 3 blocks, where each CUDA core is '*responsible*' for an element in the block. However, devices with more recent, and earlier CUDA compute capabilities do not have this 192 CUDA core / SM ratio. In fact, devices implementing CC version 6.0 have 64 cores per SM, but compensate by typically having more SMs. Take for instance, the NVidia P100, one of the latest GPUs to be produced by NVidia, which has 56 SM each with 64 cores.

(NVidia Corporation, 2016). The P100, for example, would be far better with option 1, as each SM would be able to process one block of the approximation whilst maximising achieved occupancy and capitalising the use of shared memory, seeing as each block has exclusive ownership of the shared memory at any given point. Here achieved occupancy refers to the number of active warps in a SM per clock cycle measured against the maximum number of warps per processor (NVidia Corporation, 2012). The downside of using option 1 is that it is not optimal for the current hardware, the software is being developed on. Only 64 threads are required per block, out of a potential 192, thus essentially the achieved occupancy will be low regardless of how well the algorithm performs.

5.3.1 Storage Requirements

The following section attempts to quantify the variables that need to be stored on the GPU for the SPMP3D routine to get an approximation into the required amount of shared memory. As mentioned, it is not possible to efficiently develop the OMP3D routine due to the storage requirements needed to store the Orthogonal and Biorthogonal arrays.

- The storage of three separable dictionaries with dimensions of $\mathcal{D}^x \in \mathbb{R}^{\mathbf{8\times 40}}$, $\mathcal{D}^y \in \mathbb{R}^{\mathbf{8\times 40}}$ and $\mathcal{D}^z \in \mathbb{R}^{\mathbf{3\times 15}}$. Assuming double precision storage of 8-bytes per component, $\big((8\times 40)+(8\times 40)+(3\times 15)\big)\times 8 = \mathbf{5,480}$ *bytes*.
- The need to store the image block being processed, this is simply $(8\times 8\times 3)\times 8 = \mathbf{1,536}\ bytes$.
- The need to store the residual in between iterations, this is the same size as the block being processed so $(8\times 8\times 3)\times 8 = \mathbf{1,536}\ bytes$.
- The need to store the approximation in between iterations, as with the residual this is $(8\times 8\times 3)\times 8 = \mathbf{1,536}\ bytes$.
- There is also a need to store a temporary approximation calculated at each step, this equates to $t(j)\mathbf{d}^x_{\rho^x_j} \otimes \mathbf{d}^y_{\rho^y_j} \otimes \mathbf{d}^z_{\rho^z_j}$ from Section **4.1.2.** This is also $(8\times 8\times 3)\times 8 = \mathbf{1,536}\ bytes$.
- The storage of the result of the Inner Product defined in formula (2) of Section **3.2**, which results in an array of dimensions $C \in \mathbb{R}^{\mathbf{40\times 40\times 15}}$. Clearly, $(40\times 40\times 15)\times 8 = 192{,}000$ exceeds the shared memory capacity of 48kb. As the purpose of this result is to find the maximum value and respective dictionary indices, it is possible to process each plane individually. This is discussed in more details in Section **5.3.8**. By splitting the process in this way only an array of $C \in \mathbb{R}^{\mathbf{40\times 40}}$ needs to be stored, $(40\times 40)\times 8 = \mathbf{12,800}\ bytes$, which would occupy ¼ of the shared memory. This is not a concern since this is the largest variable that needs to be stored at any point in the algorithm.
- The space required for the storage of coefficients. Since it is not possible to predetermine how many coefficients are going to be chosen from the algorithm there is requisite to pre-allocate the space. Given that fundamentally the idea is to reduce the number of atoms needed to reconstruct the image, the number of coefficients chosen will never be more than the number of components in a block. Based on this reasoning, the algorithm will pre-allocate the space required for storing the potential maximum number of chosen coefficients. Thus, $(8\times 8\times 3)\times 8 = \mathbf{1,536}$ *bytes*. It should be pointed out, that there is another approach to this. The algorithm could stop if a certain sparsity ratio

cannot be achieved. For example, if the number of coefficients exceeds a 1/3 of the number of components in a block, then break the iteration. For clarity, that is a SR is less than 3. The benefit to approaching the storage of coefficients this way is that it is then possible to pre-allocate a lower amount of space in shared memory. In this example, only space for 64 components needs to be pre-allocated reducing the amount of storage required to 512 *bytes*. Although this methodology is possible, this has not been implemented since a degree of this report is to investigate the maximum possible sparsity ratio and this would conflict with the objective.

- Additionally, there is the need to store the indices corresponding to the coefficients. Based on the same ideology just mentioned, space needs to be pre-allocated to store the maximum number of respective indices that could be chosen. That is, the size of the block. A small difference however, is that the indices will be integer type, so will only occupy 4 bytes rather than double precisions 8 bytes. Conversely, three of these arrays need to exist for the x, y and z component respectively. This therefore requires $(8 \times 8 \times 3) \times 4 \times 3 = \mathbf{2,304}$ *bytes.*

Totalling the amount of memory required for the algorithm, we get **26,728** *bytes*, which fits comfortably in the shared memory of the SM. It leaves 21.89kb of storage for temporary variables. Some of these temporary variables are still significant. For example, when computing the multiplication between three matrices, the operation is split into two steps where the output of the first multiplication is saved and used as the input to the second multiplication. This temporary variable requires **2,560** *bytes* of memory since the dimensions of the first multiplication are 8×40, so would be prudent to pre-allocate this space.

5.3.2 Memory Fragmentation

The advantage of pre-allocating space is not only limited to reducing the overhead of allocation, initialization and deallocation costs but it also lessens the occurrence of memory fragmentation. Memory fragmentation can cause severe problems, especially given there is such a small pool of memory available. During development, there were occurrences when allocation failed to return a pointer to address in memory. This is most likely due to memory fragmentation where there were no sufficiently sized spaces in memory to allocate the required resource, as demonstrated in figure 6-16.

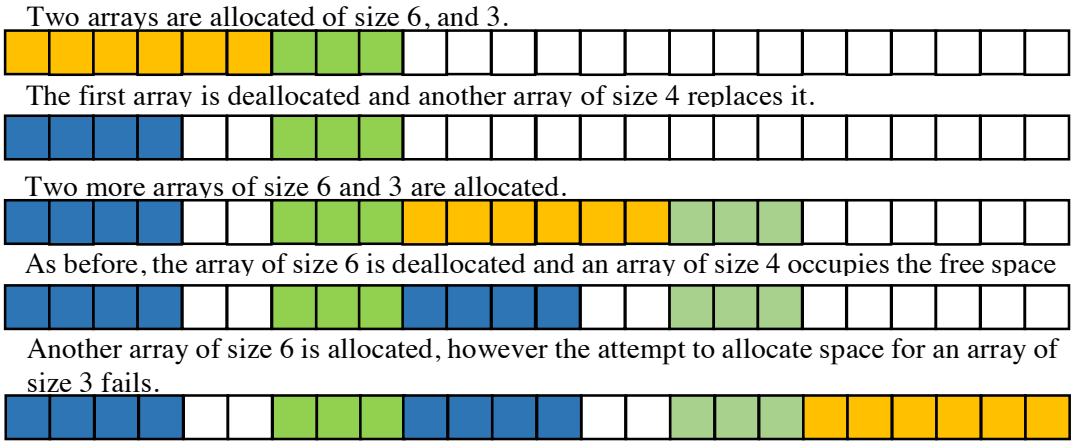


Figure 6-15 - Illustration of memory fragmentation causing a failure to allocate space

To overcome the problem of memory fragmentation the utilisation of dynamic shared memory was used for the large routine-persistent variables. This is a CUDA feature that allows you to pre-allocate shared memory on the device during the kernel launch. It is then up to the program to access the right part of this variable and cast to the desired type. An example of allocating shared memory dynamically, and accessing the variables in the kernel is shown in figure 6-17.

This approach was taken for the allocation of all the dictionaries, the image block and approximations so that they can all be allocated contiguously. In addition, any other routine-persistent variables were allocated in this way.

5.3.3 h_Matrix class

To provide a more organised structure to the program, the creation of a class 'h_Matrix' was used to encapsulate the elements and attributes of the matrix whilst also providing methods to modify and read the matrix. The class is fairly simple and only has 5 attributes: three integer types to store the height, width and depth and two double pointers, one for storing the address of the elements, and another

```
__global__ void kernelSharedDynamic(int arr1Size, int arr2Size){
	extern __shared__ char arr[];
	double* arr1 = (double*)arr;
	double* arr2 = (double*)(sizeof(double) * arr1Size + arr);
}
int main(){
	int numberOfBlocks //Can either be int or dim3
	dim3 threadsPerBlock(8,8,3); //Dimensions of the block
	int sizeOfArr1 = 100;
	int sizeOfArr2 = 200;
	int sharedMemorySize = sizeof(double) * (sizeOfArr1 + sizeOfArr2);
	kernelSharedDynamic<<<numberOfBlocks, threadsPerBlock,
sharedMemorySize>>>(sizeOfArr1, sizeOfArr2);
	return 0;
}
```

Figure 6-16 - Code example of using dynamic shared memory and accessing the memory in the kernel

for storing the address of the elements on the GPU. This second pointer is required in order to copy any resulting matrices from the GPU once the routine is complete.

The h_Matrix class is required on both the host and device, for this reason it is necessary to apply the decorators mentioned in Section **4.5.2** so that the NVCC compiler generates machine code for the GPU to understand the class. However, since the '__host__' and '__device__' type specifiers are recognised by standard C++ compilers, it was decided to define a C pre-processor macro to decorate the function if the compiler is building for CUDA. This way, the header and CPP file can be compiled with

standard C++ compiler if the application was to be expanded in the future. This of course includes refactoring the MEX implementation given time.

5.3.4 OpenCV Library

Since the GPU application is designed to be independent of MATLAB, an alternative process of loading images into the program is required. For this, the OpenCV library was used which has a C++ interface and is available on all major operating systems (OpenCV, 2017). It has trouble-free methods for loading and displaying images, this diminishes the need of having to deal with different image formats. The image is loaded into an OpenCV provided Mat object, a conversion is then applied to convert this into the previously mentioned h_Matrix object. The Mat object treats each pixel as a 3D vector with the components blue, green, and red as the elements and the h_Matrix object treats the elements like a 1D array, with column major ordering, thus in effect, the elements are flattened.

This functionality is encapsulated in a file termed 'loadImage' which shall be included when there is a need to read an image. Given a file path, the functionality contained in this module will return an object of type h_Matrix.

5.3.5 CUBLAS Library

CUDA is distributed with a GPU-accelerated implementation of the BLAS library called 'CUBLAS'. The CUBLAS library is callable from host code and manages the launching of threads to perform the operation. Since CUDA compute capability 3.5, the CUBLAS library also has a device interface so that the functions can be called from device code. When this is done the thread(s) calling the function would spawn child threads to perform the operation. However, it was decided not to use CUBLAS for this application,although there are potential performance gains to be achieved by using the library, it would also have taken away some of the control and localisation we have. When we use the CUBLAS library, we have no control over how many child threads will be spawned, and since there is a possibility that these threads are spawned on different SMs, it would affect the performance of other blocks since they may be pushed down the warp scheduler queue. No doubt, this would have worked but to avoid a chaotic design, an implementation of the Matrix Multiplication function that would have been used from the CUBLAS library has been created, and supports the same scalar and transpose operations to satisfy the below equation as defined in BLAS (NVidia Corporation, 2017). More detail of this function is shown in Section **5.3.7**.

$$C = \alpha \, op(A) \, op(B) + \beta C$$

5.3.6 CUDA Helper Functions

The creation of a file containing functions that assist with moving variables between host, device global and shared memory has been created. Notably, functions responsible for transferring h_Matrix objects between GPU and host memory were created which simplify the process considerably since it must be done in stages. Firstly, space needs to be allocated for the class container to store the integer dimensions and pointers to the element arrays. Secondly, space needs to be allocated to store the array of elements making up the matrix. Subsequently, the array of elements is copied across to the device.

Finally, the height, width, depth, and address that was returned when allocating the space on the device for the elements are copied across to populate the h_Matrix container.

The file also contains functions for extracting blocks from a large matrix; this will be used at the start of the routine for extracting the appropriate block from the image depending on the blockId of the current kernel call.

There is also a function for copying data between global and shared memory; again this will be used at the start of the routine for transferring data from the devices' global memory to the shared memory before any calculations take place. It will also be used once the routine has finished for copying the output variables from shared memory to the global memory so they can be transferred back to the host.

A notable pattern was used for this copying, more specifically a 'mapping' pattern where each thread is responsible for copying *n* elements to the new array in shared memory where:

$$n = \frac{number\ of\ elements\ to\ be\ copied}{number\ of\ threads\ in\ block}$$

This even distributes the number of elements to be copied across all the threads instead of one thread doing all the work which is the way in regular CPU programming. In scenarios where the size of the thread block doesn't fit nicely into the matrix, the functions will go one further iteration, and index checking prevents any out of bound errors.

The declaration of another pre-processor macro and inline function were used to follow CUDA guidelines of checking the return code of every CUDA function. The 'cudaCheck' macro surrounds calls to CUDA functions such as allocation and memcpy and checks the status code, if the status code does not indicate a positive result, then the program exits and the file and line number are printed out. This aided significantly whilst debugging and is based on an answer from StackOverflow (Talonmies, 2012).

A full list of the functions provided by cuda_helper.cpp is shown in figure 6-17.

Global function name	**Description**
getNumberOfSMs	Queries the number of SMs the device has.
getSmid	Queries the ID of the SM.
printDeviceDetails	Queries various other device specifications like the number of cores or CUDA version.
copyMatrixToDevice	Copies a h_Matrix object from the host memory to the GPUs global memory
copyMatrixToHost	Copies a h_Matrix object from the GPUs global memory to a matrix on the host. The source matrix must have been initialized from the host to begin with.
makeCopyOfMatrixElements	Simply makes a copy of matrix elements using a mapping pattern.
extractBlock	Extracts a submatrix from a larger matrix.
moveBetweenShared	Responsible for copying matrices between global and shared memory.

Figure 6-17 - Table containing the functions contained within the CUDA helper file.

5.3.7 Common Operations

As with the C++ MEX package, a file containing frequently used operations exists which contains functions for matrix addition, multiplication and summing the elements, to name a few. Some of these methods, for example, matrix-matrix addition are 'mappable' patterns as just discussed. Each thread sums up an equal number of elements from the two input matrices and writes to its own area in the resulting matrix.

However, some functions can't be computed in this way, for instance, the 'sum of squares' method which is used for calculating the norm $\left\|\mathbf{R}^k\right\|_{3D}^2 = \langle \mathbf{R}^k, \mathbf{R}^k \rangle_{3\mathrm{D}}$. This is because the function needs to return one value, so a scan across every element would be required. To deal with this a reduce pattern is used which is very popular in parallel programming, Firstly, each thread processes a number of equal elements each, similar to the mapping pattern, however they write the output to an array in shared memory where the thread id is the index. Once this is complete, a single thread sums up the condensed shared array and updates the output. This is very effective for large arrays, for instance it is required when searching for the maximum value after the inner product calculation. Given the matrix is $40 \times 40 = 1600$ in size, and assuming a thread block comprised of 64 threads, each thread would find the maximum value and index in a subset of 25 components. They would write the output to a shared array of size 64, before one thread finds the maximum in that shared array. By performing operations by reduction in this way, every thread is kept busy and it prevents any thread doing too much work, in this case prevents 1 thread iterating through every element which would be the way if done on CPU.

The multiplication device function 'multiplyCuda' that is a replacement for CUBLASs' dgemm is designed to work in a similar way. It accepts char arguments to tell the method whether any of the matrices need to be transposed. If it was not designed in this way, more objects would need to be created to store the transposition of the input matrices first. Depending on the combination of transposition arguments the routine will fall into 1 of four categories: Both Normal; Left Transposed, Right Normal; Left Normal, Right Transposed; Both Transposed. The only difference in these cases is the way the threads index the matrices, ultimately swapping column major for row-major ordering. The multiplication method also has three loops, two of which move the block of threads, i.e. a 'window' over the matrices being multiplied. This only occurs when the matrices exceed the thread block dimensions of 8×8. This happens in both directions hence the need for two loops. The third inevitable loop is required for summing the product of the components for each element in the resulting matrix. This method should be very efficient since it makes use of all the threads forming a block and negates the need for creating any other objects when the input matrices need to be transposed.

All the functions defined in common ops have the __device__ declaration specifier. However, there are also __global__ kernels defined which simply pass the arguments to the device function so that they can be called from the host code. This isn't currently required for any part of the application, but it assisted with testing the functions individually and allows them to be reused in the future.

A list of the functions that the common operations file provides and any GPU patterns the functions use is shown in figure 6-18.

Device function name	Global function name	Description	Notable Design Pattern
multiplyCuda	N/a	Performs C = α op (A) op (B) + β C Must use one of the matrixMultiplyCuda methods to call this.	Mapping
matrixMultiplyCuda	matrixMultiplyCudaKernel	Converts h_Matrix input arguments into double for multiplyCuda	N/a
transpose	transposeKernel	Transposes a matrix	Mapping
clearMatrix	N/a	Sets all elements to zero	Mapping
findMaxInd	findMaxIndKernel	Finds the maximum value and respective index	Reduce
squaredSum	squaredSumKernel	Performs $\left\|\mathbf{R}^k\right\|_{3D}^2 = \langle \mathbf{R}^k, \mathbf{R}^k \rangle_{3D}$	Reduce
matrixAddition	matrixAdditionKernel	Performs C = A + B	Mapping
matrixSubtraction	matrixSubtractionKernel	Performs C = A - B	Mapping
ind2sub	ind2sub	Converts an index into a set of indices (x, y, z) given some matrix dimensions.	Single Thread

Figure 6-18 - Table showing the functions contained within the common_ops file

5.3.8 Routines

This is where the more complicated algorithms are provided, for instance, an implementation of IP3D, IPSP3D, hnew, ProjMP3D and the main SPMP3D. These functions were discussed in more detail

```
#define SINGLE_THREAD if(threadIdx.x == 0 && threadIdx.y == 0)

SINGLE_THREAD{ maxVal = 0; maxInd = 0; }
__syncthreads();
```

Figure 6-19 - Code example of the SINGLE_THREAD macro

in Section **5.2**, so only differences unique to the GPU implementation shall be discussed. The routines are designed to allow all the threads to progress without divergence. However, there are blocks of code which are only performed by one thread. This is to execute tasks that only need to be performed once as to prevent race conditions i.e. initialization of shared attributes. As shown in figure 6-19, a pre-processor macro is used to define 'SINGLE_THREAD' which replaces instances with a conditional that only allows one thread to enter. This increases code readability since this is a common pattern throughout.

As discussed in Section **4.5.**4, systematically placed sync thread statements are placed throughout the routines to control flow and the alignment of threads. This is usually done after threads have amended some set of data and before other threads attempt to evaluate the variables. They are also used to make sure the threads progress through the iterations at the same rate and break from the loops at the same time.

A notable difference in the IP3D routine compared to the one programmed in C++ is that the technique for GPU calculates the inner product for each plane individually using the respective components from the 3rd dictionary. As mentioned, this reduces the requirement of having to store a matrix $C \in \mathbb{R}^{40 \times 40 \times 15}$. Instead the routine requires storage of a matrix $C \in \mathbb{R}^{40 \times 40}$ which is

calculated by performing the inner product between the image block, x dictionary and y dictionary but only one vector from the z dictionary. The maximum value and indices are stored and then the inner product is calculated again using the next vector in the z dictionary, whilst updating the maximum value if it is greater than the one currently stored. Once all the vectors in the z dictionary have been used, the selected atom will be the one that had the highest absolute maximum value. The only drawback of using this technique is the added complexity of finding the maximum at every iteration rather than just once.

All the routines have the __device__ declaration specifier since they will be predominantly called from other device code. But there are functions declared as __global __ which allow them to be put from host code for testing and reusability purposes. The exception to this is the SPMP3D kernel which doesn't merely pass the arguments to the device only function. The SPMP3D kernel expects an entire image as its argument, so it is responsible for extracting the right block and moving the dictionaries to shared memory by using the functions from 'cuda_helper.cpp'.

The full list of functions provided by the routines.cpp file is provided in table 6-20.

Function Name	**Description**
__device__ d_IPSP3d	Used in d_ProjMP3D for calculating the 3D Inner Product
__global__ d_IPSP3dKernel	Calls d_IPSP3d from host code
__device__ d_IP3d_max	Finds the indices and atom to be used in that iteration of the routine
__global__ d_IP3d_maxKernel	Calls d_IP3d_max from host code
__device__ d_hnew3d	Calculates the approximation using the selected atom
__global__ d_hnew3dKernel	Calls d_hnew3d from host code
__device__ d_ProjMP3D	See Section **5.2.5**
__global__ d_ProjMP3DKernel	Calls d_ProjMP3D from host code
__device__ d_SPMP3D	See Section **5.2.6**
__global__ d_SPMP3DKernel	Processes input arguments by moving them to shared memory, calls the main routine d_SPMP3D then frees memory once completed.

Figure 6-20 - Table showing the functions provided by routines.cpp

5.3.9 Kernel Execution

Before the kernel is executed the image needs to be loaded, dictionaries need to be generated, and space needs to be allocated on the GPU for the input and output arguments. A file 'createDicts.cpp' was created containing functions to create the dictionaries $\mathcal{D}^{x}$, $\mathcal{D}^{y}$ and $\mathcal{D}^{z}$, whilst the image is loaded using the functions provided in loadImage.cpp using the first command line argument as the target image. The space was allocated and the input arguments were moved to the GPU using the helper functions provided in cuda_helper.cpp which were discussed in Section **5.3.6**.

The dictionaries, block size and other arguments such as the max number of iterations are currently hard coded, although options to expand and customise these would be future work. The required amount of dynamically shared memory is calculated based on the size of dictionaries and block

size, but again since these variables don't currently change the amount of shared memory is also fixed and within capacity.

The number of threads per block based on the block size will be 64, more specifically a dim3 object of (8,8). Whilst the configuration for the number of blocks to launch will depend on how many partitions the image is split up into. The means in which the number of blocks, number of threads per block and amount of dynamically shared memory are calculated as well as the kernel launch are shown in figure 6-21.

```
dim3 threadsPerBlock(blockSize, blockSize,3);
dim3 numberOfBlocks(inputImage->height/8, inputImage->width/8);

d_SPMP3DKernel<<< numberOfBlocks, threadsPerBlock,(blockDim * 4 + Dx->numel() +
Dy->numel() + Dz->numel() + Dx->numel()) * sizeof(double) >>>(d_f, d_dx, d_dy,
d_dz, d_tol, d_No, d_toln, d_lstep, d_Max, d_Maxp, d_h, d_c, d_Set_ind, d_numat);

cudaCheck( cudaDeviceSynchronize() );
```

Figure 6-21 - Code example of the kernel launch, showing the kernel configuration used

A CUDA provided function 'cudaDeviceSynchronise' is called immediately after the kernel launch since normally these launches are asynchronous allowing the program to carry on. Since no other processing is completed on the CPU for this application, cudaDeviceSynchronise is called which acts as a barrier and doesn't proceed until all the blocks have been processed by the kernel. Since it is surrounded by the cudaCheck macro discussed in Section **5.3.6**, any errors during the kernel launch will be caught and displayed here. The application then proceeds to copy across output variables from the GPU to the host before freeing up any allocation before completing. The end state of the program at the current stage is that it will print out the achieved sparsity ratio and has the coefficients and set of indices ready for persistent storage. Beyond this point is not within scope for this report but shall be discussed in the future work in the conclusion.

A diagram showing the full structure of the CUDA enabled application is provided in Appendix 1, Section **10.1.3**.

5.4 Git

Throughout the project git was used for managing versions and code changes. It proved an extremely useful tool for keeping track of changes and rolling back when problems occurred. A private GitHub repository was used to allow online storage of the code base. This was very convenient since development took place on difference devices running Windows, Mac OS and Linux and the online repository kept the code up-to-date on all of the them.

GitHub was also used for GitHub pages so that the documentation can be hosted and code can be distributed in the future. Although this hasn't been completed fully, it is a baseline and can be built on as the project develops. The site can be found here: https://dannyxd11.github.io/SIA_3D/docs/, note that the repository at time of writing will be private until submission and approval.

Sparse Image Approximation in 3D

This site hosts implementations of two Matching Pursuit strategies for approximation of colour images. The strategies, termed 'Orthogonal Matching Pursuit' and 'Self-Projected Matching Pursuit' have been prototyped in MATLAB. The novelty with this project is that the algorithms attempt to take a sparse representation of every dimension similtaneously through the use of three separable dictionaries, which achieve a high redundancy whilst using low storage. Past projects have tackled the problem by vectorising images into 1D arrays or processing each colour channel individually. Ofcourse, the approach taken in this project is far more computationally intensive which is why an attempt of using the GPU for accelaration has been made.

Further C++ MEX files have been created to provide substancially increased performance, up to 90% when using the smallest realistic block size. An attempt of accelarating the process further by using NVidia's CUDA technology was also made.

The provided C++ and CUDA files were developed as part of a Final Year Project at Aston Nniversity whilst under the supervision of Dr Laura Rebollo-Neira and Dr George Vogiatzis. The MATLAB files, and the Self-Projected Matching Pursuit routine are part of a larger research programme led by Dr Rebollo-Neira, further details can be found at the projects site: http://www.nonlinear-approx.info/.

Although the software here is currently designed for Images, the same software could be used for approximation of 3D images given a suitable sized dictionary for the third / z dimension.

Licence:

MEX Files / CUDA Routine Copyright (C) 2017 Daniel Whitehouse, dannyxd@me.com, Aston University
MATLAB Routine Copyright (C) 2017, Laura Rebollo-Neira, l.rebollo-neira@aston.ac.uk, Aston

6 Evaluation

6.1 Methodology

The purpose of the evaluation is two-fold. The report first discusses two numerical tests which compare the Sparsity Ratio achieved with images when approximating each colour channel individually using existing 2D routines, and then all three colour channels simultaneously using the 3D routines. The report secondly discusses numerical tests which evaluate the performance of SPMP3D and OMP3D by using the C++ MATLAB executables. Finally, a study into the performance of MP3D is conducted with the CUDA application.

All the tests in this section were configured with a tolerance to break once the PSNR between the approximation and the original is greater than 40.5dB. This provides a satisfactory image quality and has scope for modest sparsity ratio. Figures 6-1 – 6-4 demonstrate the similarity between the original and approximation at a PNSR of 40.5dB using a block size of 8, the achieved sparsity ratio was 23.2906. The effect of block partitioning the image as discussed in **4.1.3** is imperceptible whilst looking at the picture without any magnification. However it becomes more apparent when at 10x magnification, as shown in figure 6-4.

Figure 6-1 - Original Kodak3 Image

Figure 6-2 - Approximated Kodak3 Image, Block Size 8, SR 23.2906

Figure 6-3 - Original Kodak3 Image, 10x Magnification

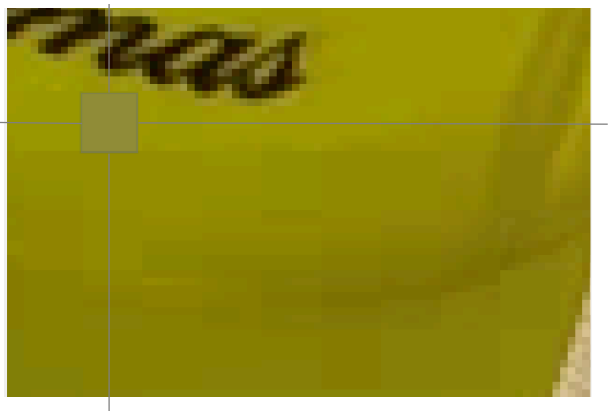


Figure 6-4 - Approximated Kodak3 Image, 10x Magnification

It is also important to note that for simplicity and to avoid the need for modification of the existing 2D routines we shall be approximating the grayscale image rather than the colour image. This can be shown to offer a very similar sparsity ratio. For clarity, table 6-1 shows the sparsity when

calculating each colour channel independently, and when approximating the image after it has been transformed to grayscale. It should be mentioned that the PSNR and SSIM index were also consistent when processing each channel individually, so only the SR is shown.

Image	Individual Channel Approximation			Average SR	Grayscale Approximation SR
	Red	Green	Blue		
Kodak1.png	4.9902	4.9535	5.0233	4.989	4.9902
Kodak4.png	5.1505	5.1324	5.2123	5.165	5.1796
Kodak8.png	5.0491	5.0236	5.0233	5.032	5.0650
Kodak12.png	11.7768	11.7466	11.6329	11.7187	11.9146

Table 6-1 - Table showing the difference in sparsity when approximating each colour channel individually and when converting to grayscale

The calculation of the PSNR for the grayscale and colour approximations is different. The grayscale is as described in Section **4.2.1**, however the MSE for the colour approximation is calculated per channel, as shown below:

$$MSE_{red} = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[\,\mathbf{I}^1(i,j) - \mathbf{A}^1(i,j)]^2\,,$$

$$MSE_{green} = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[\,\mathbf{I}^2(i,j) - \mathbf{A}^2(i,j)]^2,$$

$$MSE_{blue} = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[\,\mathbf{I}^3(i,j) - \mathbf{A}^3(i,j)]^2\,,$$

$$MSE = \frac{MSE_{red} + MSE_{green} + MSE_{blue}}{N_x \times N_y \times N_z},$$

where $\mathbf{I} \in \mathbb{R}^{N_x \times N_y \times N_z}$ is the image and $\mathbf{A} \in \mathbb{R}^{N_x \times N_y \times N_z}$ is the approximation. The PSNR is then calculated by using formula (19) from section **4.2.1**.

6.2 Comparison between approximating channels separately and simultaneously

6.2.1 Comparison between Sparsity Achieved with OMP3D and OMP2D

6.2.1.1 Aim of the numerical test

The aim of this assessment was to evaluate the sparsity gain when calculating all three colour channels simultaneously with the OMP3D routine instead of approximating each colour channel individually with the existing OMP2D routine. As of the time of writing the report no detailed comparison between the two approaches has been published, so one has been presented here.

6.2.1.2 Set up of the numerical test

The format of the test is as follows; it used a set of 16 standard sized uncompressed images (See Appendix 3) and collected the sparsity ratio achieved from OMP2D and OMP3D on varying block sizes

whilst fixing the tolerance as a PSNR tolerance value of 40.5dB. The performance of these routines in terms of time are of no interest for this test, so for uniformity, the MATLAB implementations are used. For a reasonably sized data set for comparison, 9 different block sizes were tested, ranging from 4 - 64.

In order for this test to be considered fair, the dictionaries used by each routine must have a similar redundancy to prevent bias. If the redundancy of the dictionaries is higher for one of the routines, then that routine would *possibly* be able to offer a better approximation since there are more potential atoms to be chosen.

With this in mind, for the 2D routine, we will use two dictionaries defined by $\mathcal{D}_{x_{2d}} \in \mathbb{R}^{8\times 89}$ and $\mathcal{D}_{y_{2d}} \in \mathbb{R}^{8\times 89}$, whilst for the 3D routine we will use three redundant dictionaries, $\mathcal{D}_{x_{3d}} \in \mathbb{R}^{8\times 40}$, $\mathcal{D}_{y_{3d}} \in \mathbb{R}^{8\times 40}$ and $\mathcal{D}_{z_{3d}} \in \mathbb{R}^{3\times 15}$. Calculating the redundancy of these dictionaries as the ratio between the total number of selectable atoms and the size of one block, we show that these sets of dictionaries are unbiased for comparison.

$$Redundancy\ of\ Dictionaries\ for\ OMP2D = \frac{89\times 89}{8\times 8} = 123.765625$$

$$Redundancy\ of\ Dictionaries\ for\ OMP3D = \frac{40\times 40\times 15}{8\times 8\times 3} = 125$$

6.2.1.3 Results

Table 6-2 contains the average Sparsity Ratio achieved by calculating the sparse representation of each colour channel individually, compared to calculating the sparse representation of all three colour channels simultaneously, on a sample of 16 images using OMP. Percentage Increase is included to demonstrate the relative increase in sparsity for each block size. The full set of results for this test are included in Appendix 2, Section **10.2.1**.

Block Size	4	8	16	24	32	40	48	56	64	Average
Average OMP2D Sparsity	5.438	9.215	11.67	11.66	11.92	11.81	11.89	11.93	12.07	9.54
Average OMP3D Sparsity	9.088	15.23	19.11	19.92	20.10	19.77	19.60	19.52	19.57	17.99
Percentage Increase (%)	67.11	65.38	63.78	70.88	68.59	67.32	64.93	63.52	61.84	65.93

Table 6-2 - Average Sparsity ratio achieved from a sample of 16 images on varying block sizes using the OMP algorithm

6.2.1.4 Discussion of the results

Figure 6-5 shows the sparsity of the sixteen images using the OMP3D routine; it is evident the sparsity gain of most images increases swiftly up to block sizes of 24 and then tends to level out at block size 32. From this, it can be safely determined that there is poor reason to continue increasing the block size after 24 despite the substantial increase in complexity.

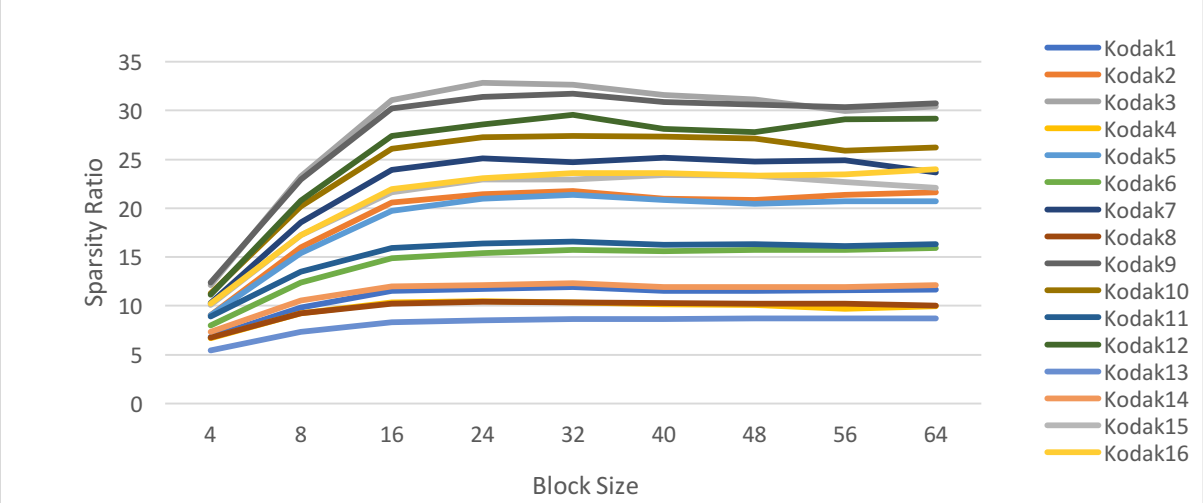


Figure 6-5- Chart showing sparsity ratio achieved with varying block sizes using the OMP3D routine

Figure 6-6 is the converse graph showing the sparsity ratio using the OMP2D routine. The main trend is visually similar however it stabilises at a lower block size of 16. Unlike the OMP3D routine where the sparsity starts to decrease after block size 32 for some images, the OMP2D routine does continue to increase, albeit very gradually. It is also important to emphasise the increase in sparsity between the routines. OMP3D creates approximations, on average, 65.93% (Fig. 6-*9)* sparser than the OMP2D counterpart, which is a significant sparsity gain. This is in line with the initial opening thoughts on this topic, where the 3D algorithms will be able to offer much sparser representations since it can make use of patterns across all three colour channels, or with 3D objects, patterns across all the dimensions.

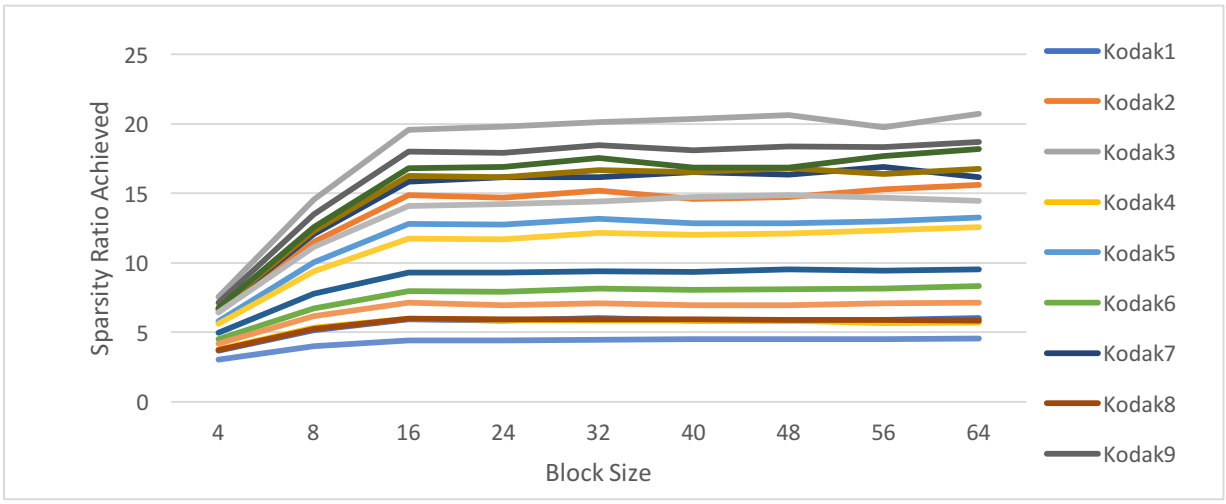


Figure 6-6 - Chart showing ratio achieved with varying block sizes using OMP2D routine

Figure 6-8 confirms the trends stated earlier, that OMP2D does continue to rise very slowly after the 'stabilising' block size of 16, whereas the OMP3D begins to decrease after reaching optimal

performance on block size 32. Since 6-7 is the standard deviation, we can also tell the spread of the data, of which it is unsurprising that the variance of the 3D routine is higher due to the increased dimensionality. It is also reasonable to see this by looking at the range of the OMP2D and OMP3D data, roughly 17 and 23 respectively. The Covariance shown in 6-8 is also affirming the statement since the variance between them is levelling out as OMP3D is decreasing and OMP2D is increasing.

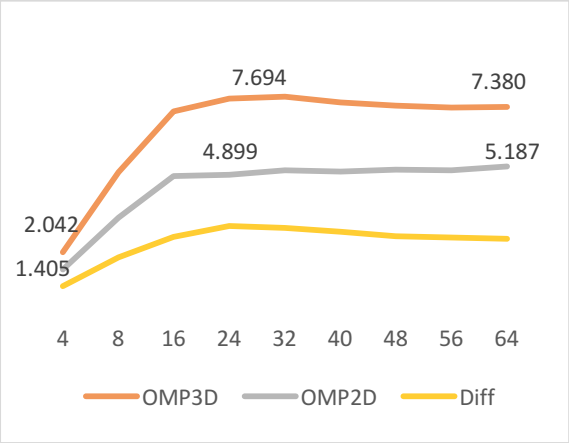


Figure 6-7 - Chart showing standard deviation of Sparsity Ratio across block sizes

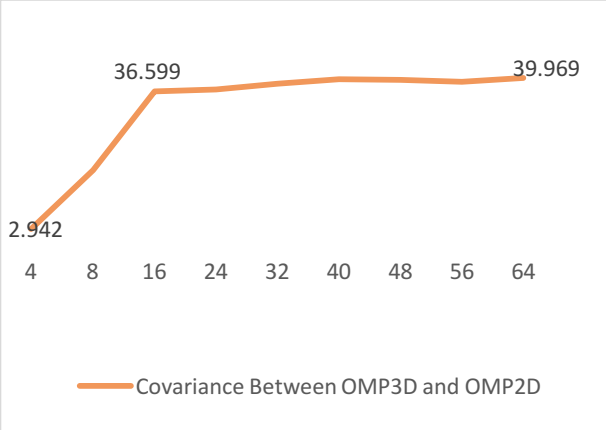


Figure 6-8 - Chart showing the covariance of Sparsity Ratio between OMP3D and OMP2D

Figure 6-9 demonstrates that on average a 65.93% increase in sparsity can be achieved by using the OMP3D routine rather than the OMP2D routine. This is a significant improvement on the OMP2D routine and has the strong potential of outperforming current image compression techniques such as JPEG2000.

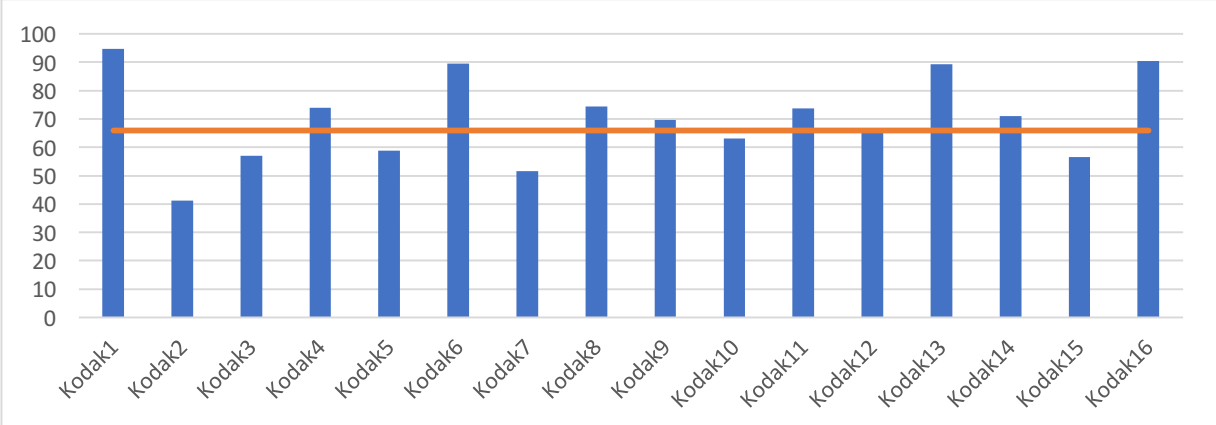


Figure 6-9- Chart showing the percentage increase in sparsity between OMP3D and OMP2D

6.2.2 Comparison between Sparsity Achieved with MP3D and MP2D

6.2.2.1 Aim of the numerical test

As with section **6.2** another test is provided to compare the sparsity gain when approximating the colour channels simultaneously instead of separately. Instead of using OMP, a particular case of SPMP is used for this test. Since when projecting at every iteration yields identical results as OMP, this test will not project at any step which is equivalent to the typical MP algorithm discussed in Section **3.4**.

Evaluation | Comparison between approximating channels separately and simultaneously

6.2.2.2 Set up of the numerical test

As with the previous test, the test will be applied on a set of 16 test images from Appendix 3. Based on the result from the previous section which showed it was redundant using larger block sizes, it was decided to use a smaller subset consisting of square block sizes between 4 and 40 for this test.

When approximating each colour separatley the dictionaries used were $\mathcal{D}_{x_{2d}} \in \mathbb{R}^{8\times 89}$ and $\mathcal{D}_{y_{2d}} \in \mathbb{R}^{8\times 89}$, whilst when approximating all channels simultaneously, the three dictionaries are defined by $\mathcal{D}_{x_{3d}} \in \mathbb{R}^{8\times 40}$, $\mathcal{D}_{y_{3d}} \in \mathbb{R}^{8\times 40}$ and $\mathcal{D}_{z_{3d}} \in \mathbb{R}^{3\times 15}$. The reason for the separate sizes are as detailed in Section **6.2.2**.

6.2.2.3 Results

Table 6-3 shows the average sparsity achieved when approximating colour channels separately and simultaneously for a set of 16 images on varying block sizes when using the MP algorithm. The percentage increase is included to show the improvement between the approaches. Full results for this test are included in Appendix 2, Section **10.2.2**.

Block Size	4	8	16	24	32	40	Average
Average MP2D Sparsity	5.12	8.06	9.88	10.32	10.49	10.35	9.04
Average MP3D Sparsity	8.19	13.47	16.67	17.33	17.40	17.06	15.022
Percentage Increase (%)	62.25	70.01	72.07	71.32	69.65	68.79	69.015

Table 6-3 - Average Sparsity ratio achieved from a sample of 16 images on varying block sizes using the MP algorithm

6.2.2.4 Discussion of the results

The overall trend is ultimately very similar when compared to OMP approach, but as expected the sparsity ratio is lower. As shown in figure 6-10, the sparsity ratio peaks with a block size of 24 for the 3D routine, with an average of 17.33, compared to the SR 10.49 at 32 blocks for the 2D routine. This is evaluated against the OMP results of 20.1 and 11.9 respectively, hence on average OMP creates approximations roughly 15% sparser than the conventional matching pursuit algorithm.

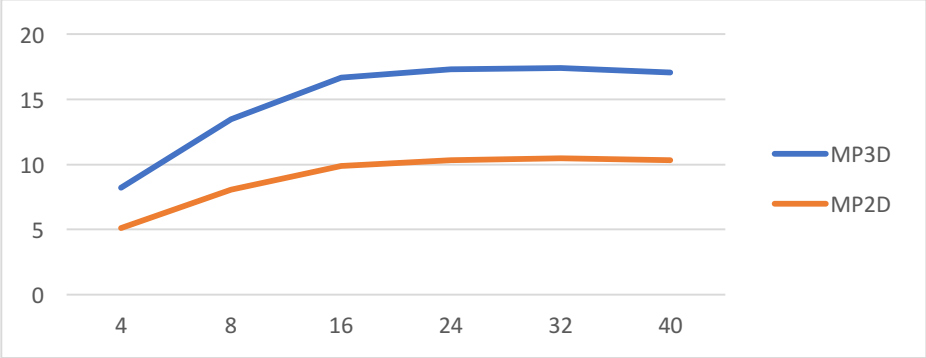


Figure 6-10 - Chart showing the average sparsity ratio of MP3D vs MP2D

Evaluation | Comparison between approximating channels separately and simultaneously

The percentage increase in sparsity is consistent with the results found in Section **6.2**, with an average increase of 69%. However, the standard deviation of this result is 13.203, so the data is quite spread and depends on the maximal possible sparsity for that image. A plot of the percent increase for each image for varying block sizes is shown in figure 6-11, with the average line 69% being also plotted.

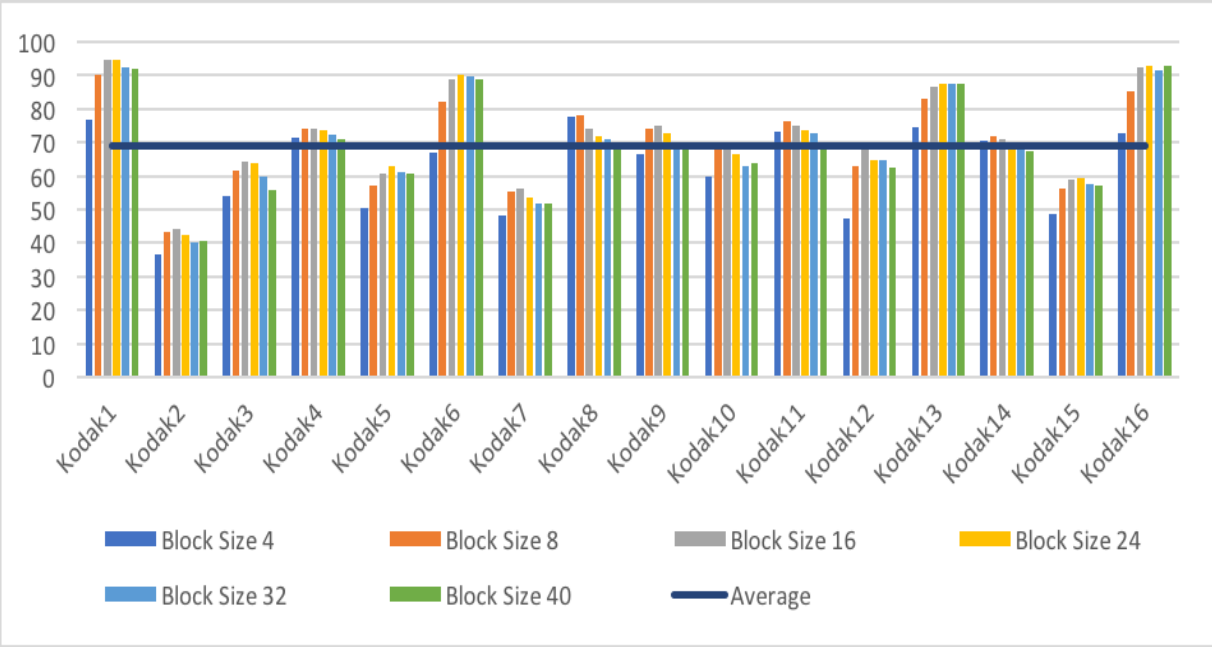


Figure 6-11 - Chart showing the Percentage Increase in Sparsity across various Block Sizes

6.3 Performance Comparison of MEX files

The platform that was used for the execution of the tests in Sections **6.3** and **6.4** had the following specifications:

Intel i7-3770, 4 Cores X 3.4 GHz, 8MB Cache

16GB DDR3 RAM

Seagate 1TB HDD 7200RPM

NVIDIA GTX TITAN 6GB, 2688 CUDA Cores

OS: Ubuntu 14.04

6.3.1 Orthogonal Matching Pursuit 3D

6.3.1.1 Aim of the numerical test

The purpose of this test is to evaluate the speed increase when using the C++ MEX routine rather than the standard MATLAB implementation.

6.3.1.2 Set up of the numerical test

Both the C++ MEX version and the MATLAB version of the routine were used to approximate each image of the test set provided in Appendix 3. This was performed three times to get an accurate and reliable result. For OMP3D, only block sizes 4 and 8 were considered since for larger block sizes it is beneficial to use SPMP3D because of the lower memory requirement.

The routine was triggered to stop once the approximation had achieved a PSNR value above 40.5dB, and since the MATLAB and C++ MEX routines perform exactly the same operation, it is expected that both MEX and MATLAB return the same Sparsity Ratio, PSNR and SSIM index.

6.3.1.3 Results

Figure 6-12 shows the average execution time of approximating the 16 images in the test set using the MATLAB routine and C++ MEX routine on two different square block sizes: 4 and 8. As expected the PSNR, SSIM and SR were identical between the two routines. The full set of results are included in Appendix 2, Section **10.2.3**.

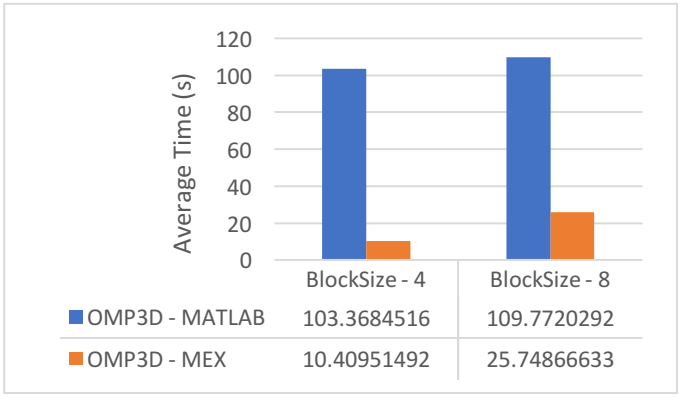


Figure 6-12 - Chart showing the average time to complete the OMP3D routine using MATLAB and MEX

6.3.1.4 Discussion of the results

Studying Figure 6-12, as anticipated, the execution time does increase as the block size increases. However the MEX routine seems more sensitive to the change in block size. The routine took 15 seconds longer, 150% increase when using a block size of 8, whereas the MATLAB routine only increased by 6 seconds. This could, however, be because MATLAB's vectorization advantage does not help with the 4-block routine because the compute complexity is so small anyway. As such, it follows that as the block size increased to 8 MATLAB performs better i.e. the increase in execution time is smaller to that of the MEX routine.

Figure 6-13 illustrates the percentage difference between the MATLAB routine and MEX routine. It is clear that it is very advantageous to use the MEX version as it can reduce the execution time

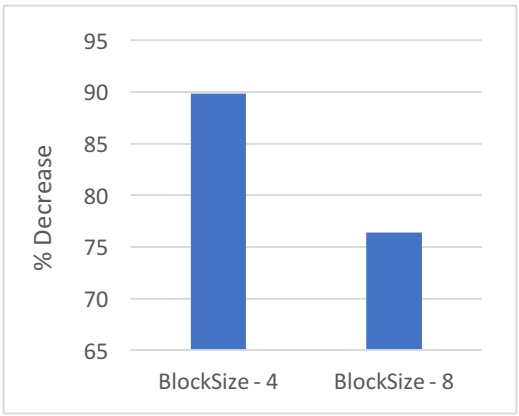


Figure 6-13 - Chart showing the average percent decrease in execution time between MATLAB and MEX

significantly;- up to 90% on the smallest possible block size. The standard deviation of the 4-block percentage decrease was $\sigma = 0.2909$, indicating very little spread, so the 89.8% decrease in execution time was consistent across all the test images. Whereas the standard deviation of the 8-block percentage decrease was $\sigma = 2.736$, slightly higher but nonetheless, the range of deduction was between 70%-80%.

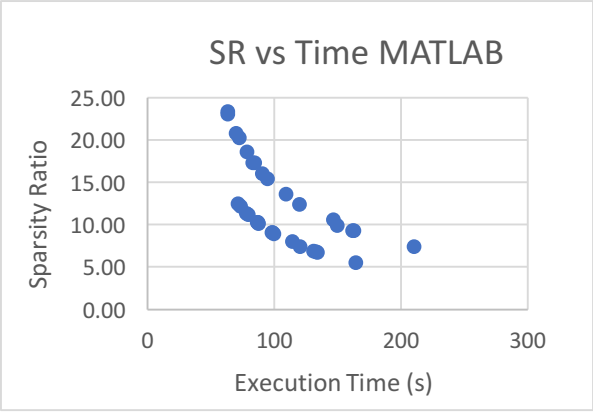


Figure 6-14 - Chart showing the relationship between SR and the time taken with MATLAB

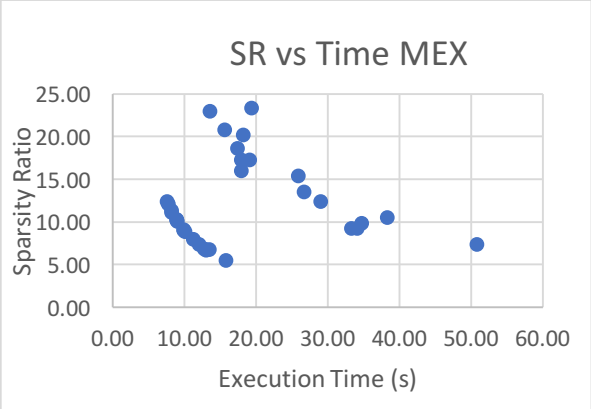


Figure 6-15 - Chart showing the relationship between SR and the time taken with MEX

Another interesting observation is the relationship between the execution time of the routine and the achieved sparsity. The two scatter graphs shown in Figures 6-14 and 6-15, comparing the SR and Execution Time of the MATLAB and MEX routines respectively indicate a negative trend. That is, the longer it takes for the routine to approximate the image, the poorer the SR. This is sound logic since the routine is having to perform more iterations to meet the predefined tolerance levels, as such the number of coefficients being selected also increases. Notice also the irregularity in figure 6-15 with the top curve; this shows the MEX routine is less consistent than MATLAB with larger block sizes and is possibly why the standard deviation is higher.

6.3.2 Self-Projected Matching Pursuit 3D

6.3.2.1 Aim of the numerical test

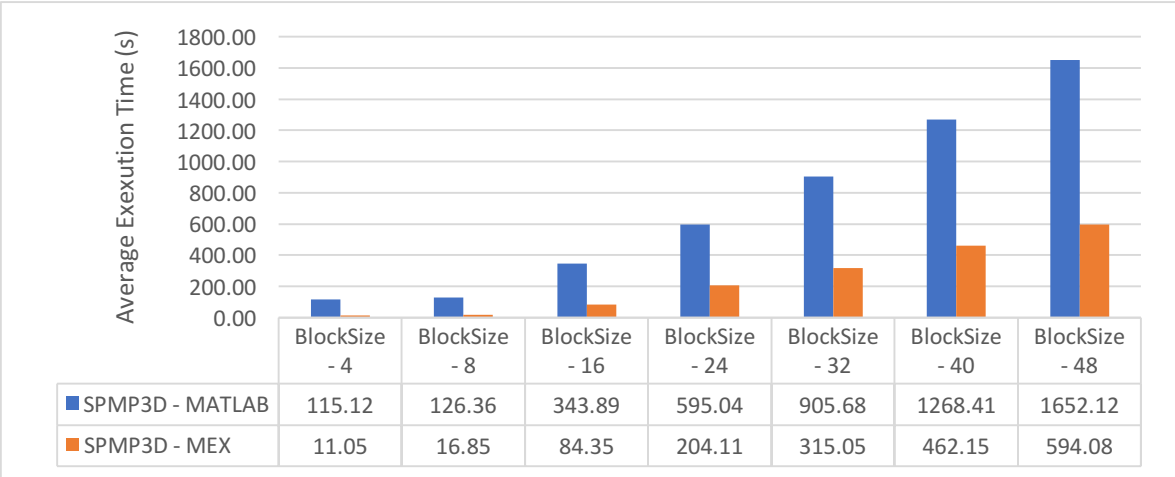
As with Section **6.3.1,** the purpose of this experiment is to access the performance increase using the SPMP3D routine developed in C++ compared the MATLAB implementation.

6.3.2.2 Set up of the numerical test

The test was performed using seven different block sizes going up to 48. This is possible as the SPMP routine eliminates the need to store the large orthogonal arrays like in the OMP routine. To reduce execution time only one run was performed on each image, instead of three like in the OMP test. However, to ensure there were no anomalies, the trends were compared to the time taken in the OMP routine since they should be relatively consistent.

6.3.2.3 Results

Figure 6-16 illustrates the average execution time for the MATLAB and C++ MEX routines on a series of images on different block sizes. The SR, PSNR and SSIM were identical between the two approaches, so we can be confident that the C++ implementation is working as expected. The full results for this test are shown in Appendix 2, Section **10.2.4**.



	BlockSize - 4	BlockSize - 8	BlockSize - 16	BlockSize - 24	BlockSize - 32	BlockSize - 40	BlockSize - 48
SPMP3D - MATLAB	115.12	126.36	343.89	595.04	905.68	1268.41	1652.12
SPMP3D - MEX	11.05	16.85	84.35	204.11	315.05	462.15	594.08

Figure 6-16 - Comparison of average execution time between MATLAB and C++ MEX implementations for varying block sizes

6.3.2.4 Discussion of the results

Since this test was performed on a larger number of different block sizes, it is apparent from Figure 6-17 that the increase in performance converges at around 65% for large block sizes. This signifies that the algorithm in C++ is around three times quicker for block sizes of 24 or greater. When the block size is smaller the performance is further improved, up to a 90% reduction in execution time when the block size is 4.

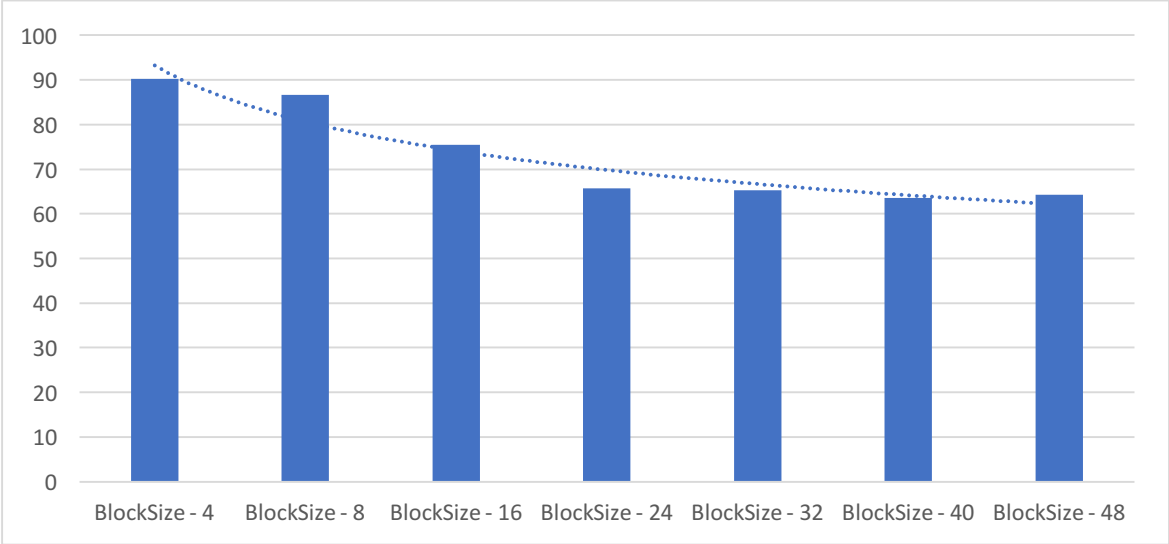


Figure 6-17 - Plot of Average Percentage Decrease in execution time between MATLAB and C++ MEX

Although only a small sample was taken for OMP3D, the increase in performance for SPMP3D seems slightly better than that achieved in OMP3D. When the block size is 8, the performance increase for SPMP3D is 85% compared to OMP3Ds 76%. This could be explained by SPMP3D consisting of

more loop type structures due to the Projection stage. Thus since C++ is superior with loops compared to MATLAB, there is more to improve in SPMP3D.

Figure 6-18 shows the execution time and the achieved sparsity for the images in the test set. Each point in the trend lines represents a block size between 8-48. It is evident from the graph, as found in the previous sections that there is little point using block sizes greater than 24 since the sparsity ratio does not continue to improve. However, this graph also shows the considerable execution time and shows that despite no increase in sparsity, there is on average a 2.5x increase in execution time between using block sizes 24 and 48.

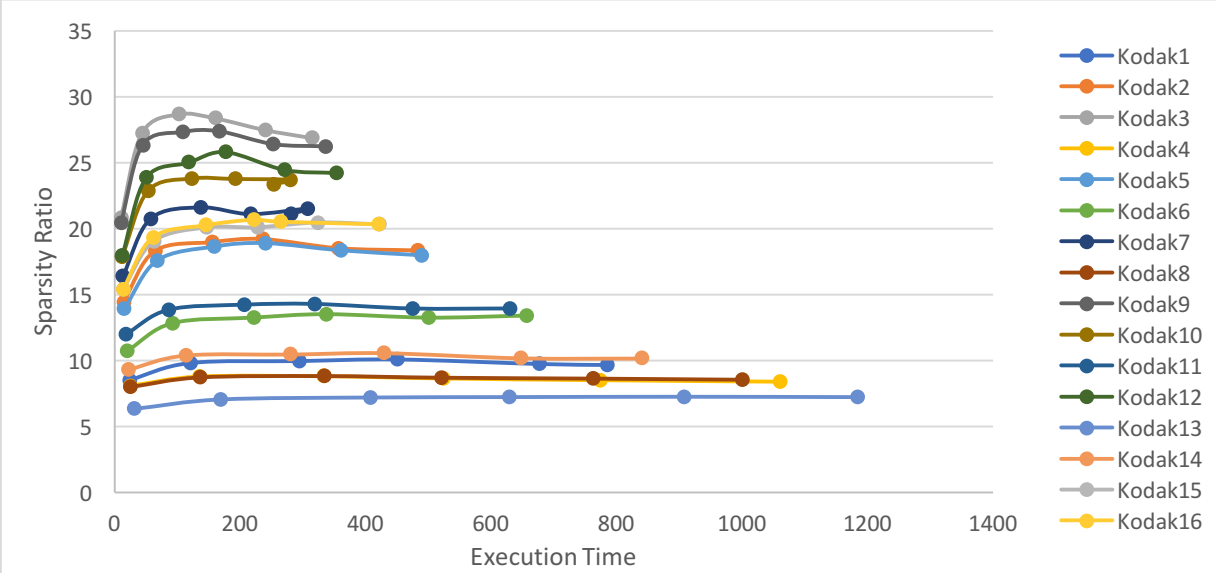


Figure 6-18 - Chart showing the execution time and sparsity ratio for each image on varying blocks

6.4 Performance Comparison of CUDA application

6.4.1 Aim of the numerical test

The purpose of this test is to evaluate the performance of the GPU application in comparison to the MATLAB routine. Due to time constraints, only the Matching Pursuit algorithm was completely realisied for the GPU application. However, the foundations of the projection step have been created for future use in the SPMP algorithm.

6.4.2 Set up of the numerical test

The format of the tests is as follows, a number of images of increasing pixel count were approximated using the MATLAB routine and the GPU implementation. The target PSNR is 40.5dB, whilst using a block size of 8 x 8. This is due to memory limitations of the GPU as discussed in Section **5.3**. Currently the configuration is hardcoded in the GPU application but this can easily be extended in future work.

6.4.3 Results

Table 6-4 contains the results from a small test set on the GPU. Five images of increasing pixel count were used to investigate the performance of the CUDA application.

Image Size	**256 X 256**	**512 X 768**	**800 X 1280**	**1731 X 1280**	**2560 X 1688**
Pixel Count	65536	393216	1024000	2211840	4321280
No. Coefficients	9602	139676	353328	275725	1015012
Sparsity Ratio	20.476	8.446	8.694	24.066	12.772
MATLAB (s)	11.959	195.999	484.847	356.319	2591.000
C++ MEX (s)	1.735	22.318	58.684	47.456	167.799
CUDA Application (s)	1.279	13.983	50.280	36.568	121.731
MEX Improvement (%)	85.5%	88.6%	87.9%	86.7%	93.5%
CUDA Improvement (%)	89.3%	92.9%	89.6%	89.7%	95.3%

Table 6-4 - Table containing the comparison between the CUDA application, MATLAB and C++ MEX

6.4.4 Discussion of the results

Figure 6-19 illustrates the execution time compared against image size measured in pixel count. We can see that the CUDA application slightly outperforms the C++ MEX implementation, despite the BLAS optimisation that was utilised. The percentage increase is very stable in the chosen subset of images with standard deviation of $\sigma = 0.024$.

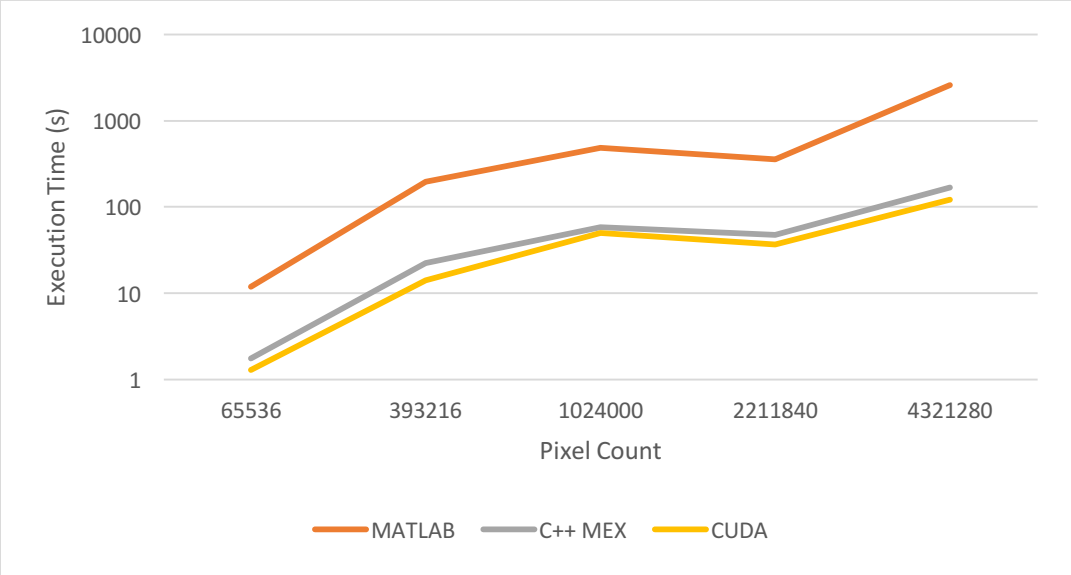


Figure 6-19 - Chart showing the execution time and pixel count of the images when approximated with MATLAB, C++ MEX and CUDA

An observation was also made between the sparsity ratio and the difference between the C++ MEX performance the CUDA performance. It is conjectured, that images that achieve a higher sparsity ratio require fewer iterations. Since the clock speed on the GPU is slower than a CPU, the GPU will perform better when fewer iterations for one block are required. Although the sample is small, a gradual upward trend shown in Figure 6-20 was observed. It would have been interesting to investigate this further with a larger subset of images, but due to time constraints this was not possible

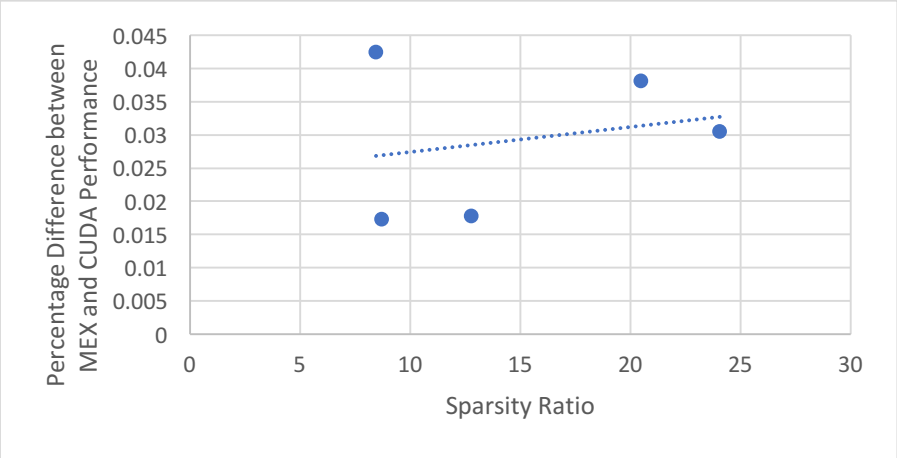


Figure 6-20 - Chart showing the relationship between the Sparsity Ratio and performance difference between C++ and CUDA

It should be noted that the results displayed here are on a GPU whose theoretical achieved occupancy is a dwindling 3%, which was derived from the occupancy calculator (NVidia Corporation, 2017). This is because the CUDA Compute Capability version of the hardware is 3.5, which indicates the GPU has 192 ALU lanes / CUDA cores. The application is only utilising 64 threads per block, and occupying enough shared memory that no more than one block can be launched simultaneously on one SM. Taking this into account, the hardware being used can approximate 14 blocks in parallel.

It is unfortunate that there are these limitations on the test hardware, however, lets speculate on the outcome of newer GPUs, for instance, a high range Tesla P40 which is CUDA CC version 6.1. This implies each SM has 96kb of shared memory available and 128 CUDA Cores, so it would be possible to run 3 blocks per SM. The GPU also benefits from having a larger number of SMs; the Tesla P40 for instance has 60 SMs available. This would allow an outstanding 180 blocks to be run in parallel, which is 12 times more than the current hardware. Not only that, newer GPUs also benefit from having a higher clock rate and smaller transfer speeds (NVidia Corporation, 2017).

Another approach that could be taken is the utilisation of multiple GPUs. The CUDA runtime API automatically picks up any CUDA capable devices and distributes blocks to be calculated over all the devices. This could turn out to be a more cost-effective solution.

7 Conclusion

Two distinct applications have been created as a result of this report. Firstly, a C++ application was presented to decrease the execution time which utilises optimisation libraries, such as BLAS. From examination, the C++ implementation of both algorithms was proven to drastically increase performance within the range of 65-90% for suitably sized image partitions.

The concept of accelerating the approximations with the GPU has been reviewed, and a study into the storage requirements needed for the routines found that the GPUs shared memory is capable of storing the required variables. Utilisation of this fast-access memory can reduce read and write latency due to the iterative nature of the algorithms.

A GPU application was created that currently implements the MP3D algorithm. The basis of the projection stage has been outlined giving the possibility of fully implementing the SPMP3D algorithm in further developments. As discussed in the evaluation, when combined with more powerful or multiple devices, the GPU could significantly increase the performance of these 3D approximations.

A detailed investigation was conducted into the gain in sparsity achieved by simultaneously approximating three colour channel images, instead of the previously discussed 2D approximations of each channel. This investigation found there was on average a significant 66% increase in sparsity over a standard set of sixteen test images.

7.1 Reflection

The entire project was very fulfilling, although the author had some prior experience with programming languages, he was able to expand this area of knowledge to include C++ and CUDA. This does not mention all the complimentary tools that were studied and utilised throughout to aid development, testing and debugging. For instance, Git versioning and application profiling.

The author was able to develop an in-depth understanding of how Sparse Image Approximation is performed and the mathematical basis on which it is grounded. This project has involved an area of applied mathematics in which the author had no previous experience, but throughout the duration of the project has developed a keen interest, and may even continue to improve the application after submission. It has been fascinating to see the considerable gain in sparsity as a result of novel approximation in 3D.

The author is delighted with the outcome of the CUDA application as this was an ambitious and extremely difficult task. The author built a firm understanding of GPU architecture and parallel programming concepts. The ever so slight performance increase is captivating considering the test hardware isn't the most suitable for the task, and the author is confident that with multiple GPUs, or more powerful GPUs there would be a far more substantial difference between the CUDA and C++ routines. On the whole, the non-trivial concept of GPU programming for acceleration is an exciting area of Computer Science and another area which has caught the author's long-term interest.

7.2 Future Work

There are a few directions the project could take to evolve. Firstly, a method of storing the coefficients and indices in a file so that they can be moved around and reconstructed at another location. As already mentioned, the work discussed in the report is the first step in a transformation based compression scheme. Thus, before saving to a persistent storage medium, there would be further stages to reduce the amount of space that would be required. This task in itself would involve entropy encoding techniques, like Huffman encoding and run-length encoding to ensure the file size is as small as possible.

The C++ MEX files could be adapted so that it is independent of MATLAB, similar to the GPU application. By doing this, it would allow the software to be used by a wider audience since there would be no dependency on the end-users having MATLAB. The program currently has a reliance on MATLAB for providing the input parameters to the routine, dealing with the image partitioning and iterating through blocks, and providing a hassle-free interface to a BLAS library. Providing the input parameters can be solved in a similar way to the GPU by using the OpenCV library to handle the loading of Images. Image partitioning is a relatively trivial task and could be further accelerated by using C++ since it is implied from this report it is quicker when dealing with loop blocks. It may also be possible to look at multi-threading to spread the complexity over multiple cores. The MATLAB proprietary BLAS interface could be replaced with an open source implementation such as OpenBLAS (OpenBLAS, 2017) which from exploratory research would require minimal code change.

All of the test subjects in this report have been colour images, where the 3D aspect is made possible by evaluating the red, green and blue colour channels together. For further investigation into sparsity gain, it would be easy to expand the algorithms to support objects with a larger number of planes in the Z-dimension.

8 References

Al-Nuaim, H., & Abukhodair, N. (2011, November). A User Perceived Quality Assessment of Lossy Compressed Images. *International Journal of Computer Graphics, 2*(2).

Bourque, B. (2014, April). *The differences between a GIF, a JPG, and a PNG explained*. Retrieved March 2017, from http://www.digitaltrends.com/computing/whats-the-difference-between-a-gif-a-jpg-and-a-png-file/

Datta, R., Joshi, D., Li, J., & Wang, J. Z. (2008, April). *Image Retrieval: Ideas, Influences, and Trends of the New Age*. Retrieved March 2017, from http:// doi.acm.org/10.1145/1348246.1348248

Dongarra, J., Duff, I., Croz, J. D., & Hammarling, S. (1989). *Subroutine DGEMM*. Retrieved from netlib.org./blas: http://www.netlib.org/blas/dgemm.f

Harris, M. (2013, Feb). *An Efficient Matrix Transpose in CUDA C/C++*. Retrieved from NVidia Blogs: https://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/

Harris, M. (2013, Janurary 28). *Using Shared Memory in CUDA C/C++*. (NVidia) Retrieved March 2017, from NVidia Blogs: https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/

Huynh-Thu, Q., Garcia, M.-N., Speranza, F., Corriveau, P., & Raake, A. (2011, March). Study of Rating Scales for Subjective Quality Assessment of High-Definition Video. *IEEE TRANSACTIONS ON BROADCASTING,, 57*(01), 1.

J. Klaue, B. R. (2003, Sept). “EvalVid - a framework for video transaction and quality evaluation. *13th Intl. Conference on Modelling Techniques and Tools for Computer Performance Evaluation*.

Joint Photographers Expert Group. (2000, Jan). *Overview of JPEG 2000*. Retrieved March 2017, from https://jpeg.org/jpeg2000/index.html

Kaur, R., & Choudhary, P. (2016, May). A Review of Image Compression Techniques. *International Journal of Computer Applications, 142*(1).

Khronos Group. (2017). *OpenCL*. Retrieved from Khronos Group - Connecting Software to Silicon: https://www.khronos.org/opencl/

King, L. (2013, August). *IBM Big Data & Analytics Hub*. Retrieved March 2017, from http://www.ibmbigdatahub.com/blog/why-compression-must-big-data-era

Kodak. (2010). *Kodak Lossless True Color Image Suite*. Retrieved March 2017, from PhotoCD PCD0992: http://r0k.us/graphics/kodak/

Krishnan, S. (2014, May). *Using Compression Techniques to Streamline Image and Video Storage and Retrieval*. (Cognizant) Retrieved March 2017, from https://www.cognizant.com/industries-resources/media_and_entertainment/Using-Compression-Techniques-to-Streamline-Image-and-Video-Storage-and-Retrieval.pdf

Lawson, C. L., Hanson, R. J., Kincaid, D. R., & Krough, F. T. (1979). Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 308-323.

Mallat, S., & Zhang, Z. (1993). Matching pursuit with time-frequency dictionaries. *IEEE Transactions on Signal Processing, 41*(12), 3397–3415.

Mathworks. (2017). *The Language of Technical Computing*. Retrieved from MATLAB: https://www.mathworks.com/products/matlab.html

Michigan Library, U. o. (2017, March 29). *Research Guides*. Retrieved March 2017, from http://guides.lib.umich.edu/c.php?g=282942&p=1885348

NVidia Corporation. (2012). *Cuda Profile Settings*. Retrieved March 2017, from NVidia Nsight Visual Studio Edition: http://http.developer.nvidia.com/NsightVisualStudio/2.2/Documentation/UserGuide/HTML/Content/Profile_CUDA_Settings.htm

NVidia Corporation. (2016). *NVIDIA Tesla P100*. Retrieved from NVidia Whitepaper: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

NVidia Corporation. (2017). *CUBLAS Guide*. Retrieved from CUDA Toolkit Documentation: http://docs.nvidia.com/cuda/cuda-samples/index.html#simple-cublas

NVidia Corporation. (2017). *CUDA C Programming Guide*. Retrieved from NVidia Docs: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

NVidia Corporation. (2017). *Occupancy Calculator*. Retrieved from CUDA Occupancy calculator.xls - Nvidia

NVidia Corporation. (2017). *Tesla P40 Data Sheet*. Retrieved from NVidia: https://www.microway.com/download/datasheet/NVIDIA_Tesla_P40_Data_Sheet.pdf

OpenBLAS. (2017). *An optimised BLAS library*. Retrieved from OpenBLAS: http://www.openblas.net/

OpenCV. (2017). *OpenCV*. Retrieved from http://opencv.org/

Pati, Y., Rezaiifar, R., & Krishnaprasad, P. (1993). Orthogonal matching pursuit: recursive function approximation with applications to wavelet decomposition. *Proc. of the 27th ACSSC, 1*, 40-44.

Patil, S., & Sheelvant, S. (2013). Survey on Image Quality Assessment Techniques. *International Journal of Science and Research*.

Rebollo-Neira, L. (2017). Notes on Greedy Strategies for Sparse 3D Image Approximation. *Unpublished*.

Rebollo-Neira, L., & J.Bowley. (2013). Sparse representation of astronomical images. *Journal of The Optical Society of America A, 30*, 758-768.

Rebollo-Neira, L., & Lowe, D. (2002, April). Optimized Orthogonal Matching Pursuit Approach. *IEEE SIGNAL PROCESSING LETTERS, 9*(4), 137.

Rebollo-Neira, L., & Sasmal, P. (2016). *Low memory implementation of Orthogonal Matching Pursuit like greedy algorithms: Analysis and Applications*. Retrieved from http://arxiv.org/abs/1609.00053

Rebollo-Neira, L., Bowley, J., Constantinides, A., & Plastino, A. (2012). Self contained encrypted image folding. *391*, 5858–5870.

Sattigeri, P., Thiagarajan, J. J., Ramamurthy, K. N., & Spanias, A. (2012, Jan). IMPLEMENTATION OF A FAST IMAGE CODING AND RETRIEVAL SYSTEM USING A GPU. *Emerging Signal Processing Applications (ESPA)*.

Talonmies. (2012, December). *What is the canonical way to check for errors using the CUDA runtime API?* . Retrieved from StackOverflow: http://stackoverflow.com/questions/14038589/what-is-the-canonical-way-to-check-for-errors-using-the-cuda-runtime-api

Tvrdik, P. (1999, Jan 21). *PRAM Models*. Retrieved from http://pages.cs.wisc.edu/~tvrdik/2/html/Section2.html

VQEG. (2000, March 1). *Final report from the video quality experts group on the validation of objective models of video quality assessment*. Retrieved March 4, 2017, from http://www.vqeg.org/

Wang, Z., C. Bovik, A., & Lu, L. (2002). *Why is image quality assessment so difficult?* (D. o. Lab for Image and Video Engi., Ed.) Retrieved March 04, 2017, from http://live.ece.utexas.edu/publications/2002/zw_icassp2002_whyqa.pdf

Wang, Z., Conrad Bovik, A., Rahim Sheikh, H., & P. Simoncelli,, E. (2004, April). Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE TRANSACTIONS ON IMAGE PROCESSING, 13*(4), 600-612.

Yogita V, H., & Y. Patil,, H. (2015). A Survey on Image Quality Assessment Techniques, Challenges and Databases. (N. C. Computing, Ed.) *International Journal of Computer Applications*.

9 Bibliography

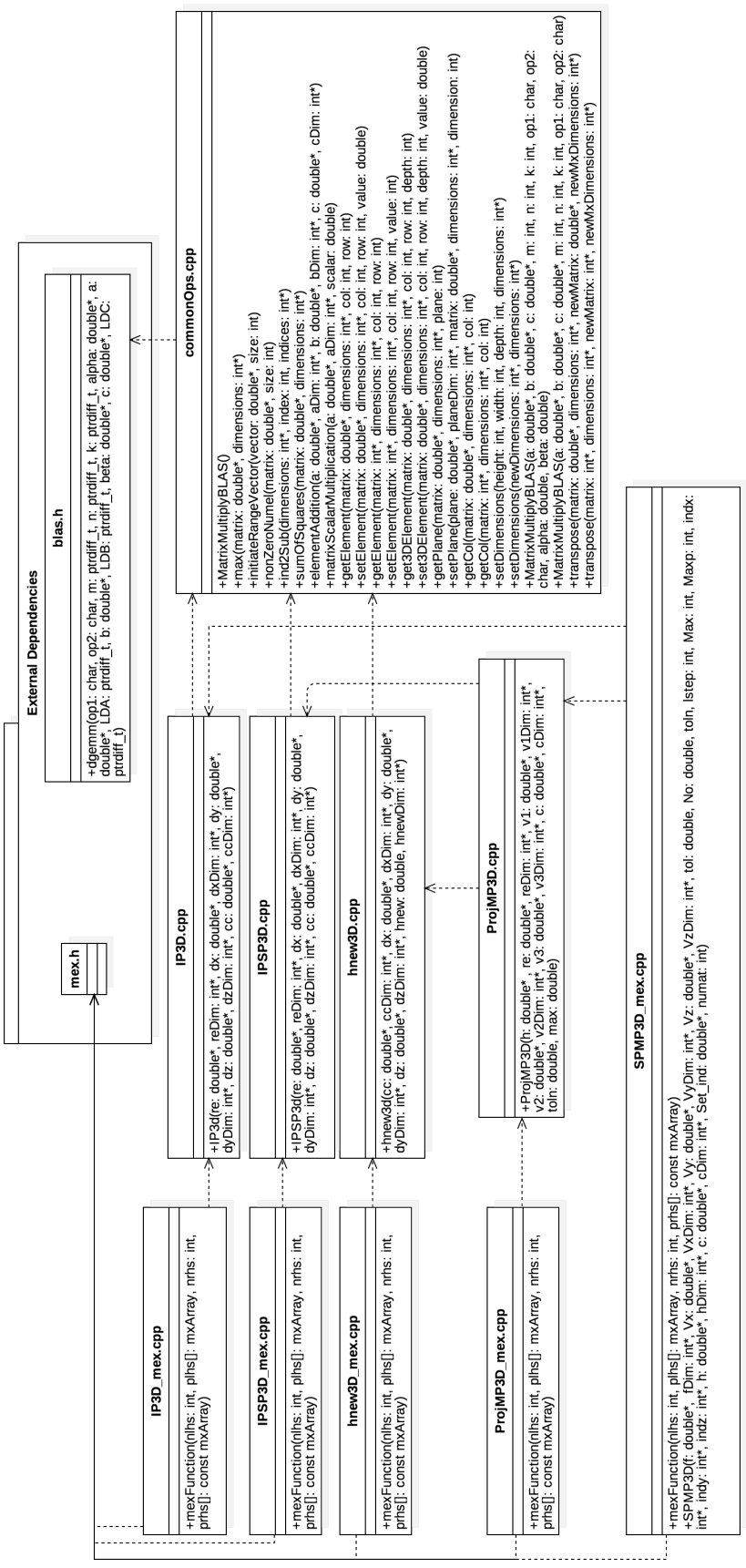
Suh, Jung W., and Youngmin Kim. Accelerating MATLAB with GPU Computing: A Primer with Examples. Waltham, MA: Elsevier/Morgan Kaufmann, 2014. Print.

Simons, Matt. Orthogonal Matching Pursuit in 2D for Java with GPGPU Prospectives. Rep. Aston University, 6 May 2014. Web. <http://www.aston.ac.uk/study/undergraduate/courses/eas/bsc-mathematics/matt/>.

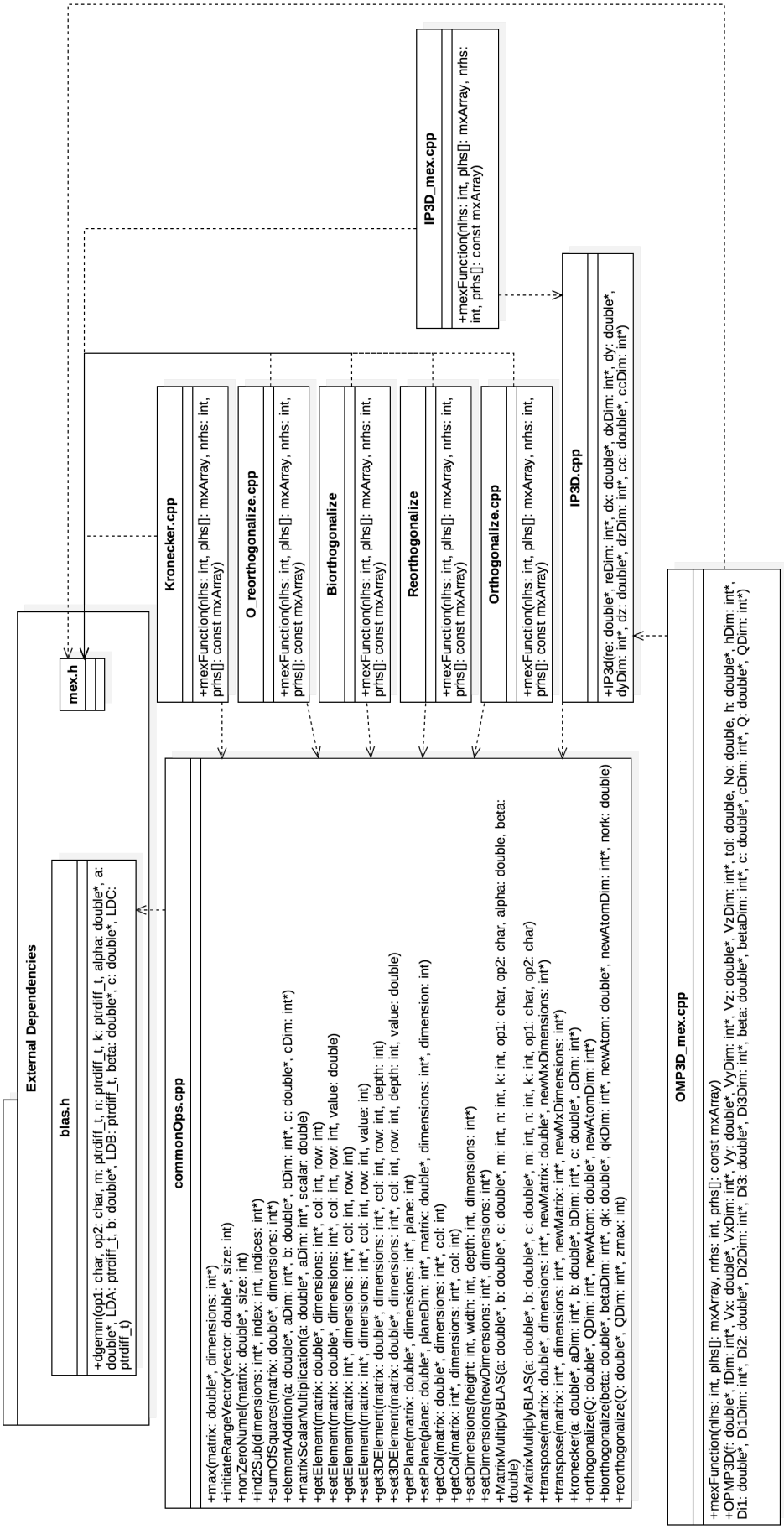
10 Appendices

10.1 Appendix 1 – Dependency Diagrams

10.1.1 Dependency diagram for SPMP3D routine in MEX C++



10.1.2 Dependency diagram for OMP3D routine in MEX C++



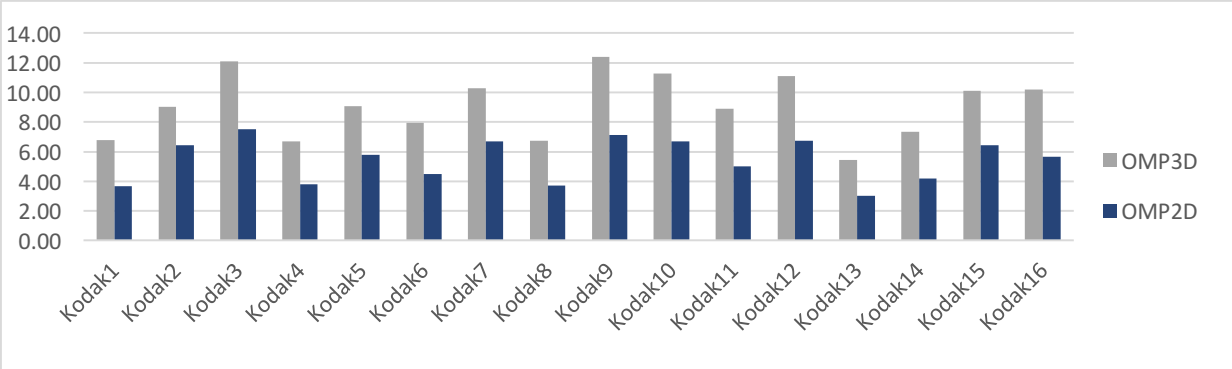
10.1.3 Dependency diagram for the CUDA Application

10.2 Appendix 2 - Detailed Results

10.2.1 Detailed results for Sparsity Achieved with OMP3D and OMP2D

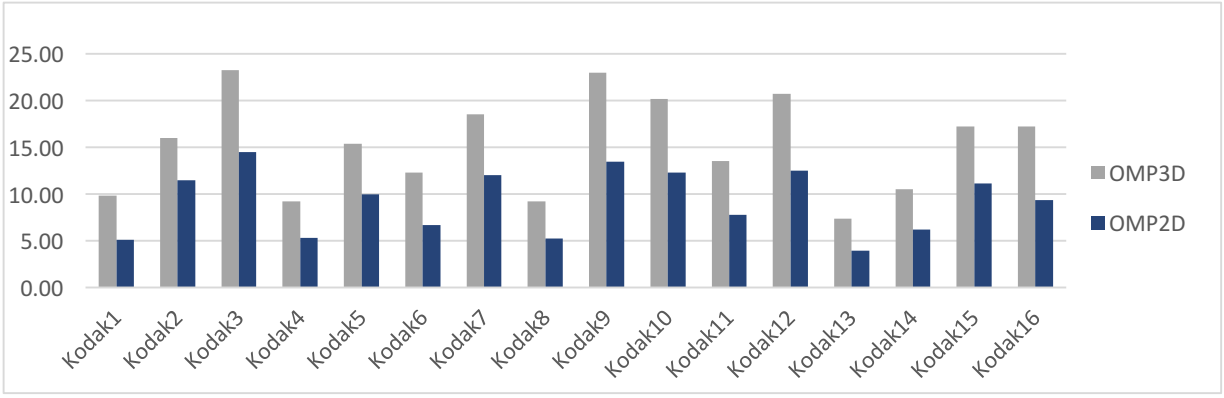
10.2.1.1 Block Size 4

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
OMP3D																		
PSNR	41.82	42.06	42.73	41.68	41.93	42.05	42.65	41.62	41.92	41.93	42.06	42.29	41.59	41.71	42.35	42.20		
Sparsity Ratio	6.80	9.01	12.10	6.68	9.06	7.96	10.27	6.75	12.40	11.29	8.89	11.12	5.44	7.36	10.09	10.21	9.088	2.042
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.93	0.96	0.90	0.99	0.99	0.98	0.98	0.98		
OMP2D																		
PSNR	42.76	43.04	43.97	42.79	42.89	43.10	43.91	42.59	42.73	42.81	43.13	43.33	42.62	42.60	43.66	43.24		
Sparsity Ratio	3.67	6.44	7.53	3.80	5.81	4.50	6.69	3.71	7.14	6.70	4.99	6.76	3.03	4.19	6.43	5.64	5.438	1.405
SSIM	0.95	0.78	0.70	0.97	0.84	0.85	0.75	0.96	0.62	0.74	0.85	0.74	0.98	0.95	0.71	0.82		
% Increase	85.41	39.96	60.62	75.72	56.08	77.00	53.59	81.91	73.56	68.60	78.13	64.52	79.76	75.57	56.86	80.88	69.26	12.33



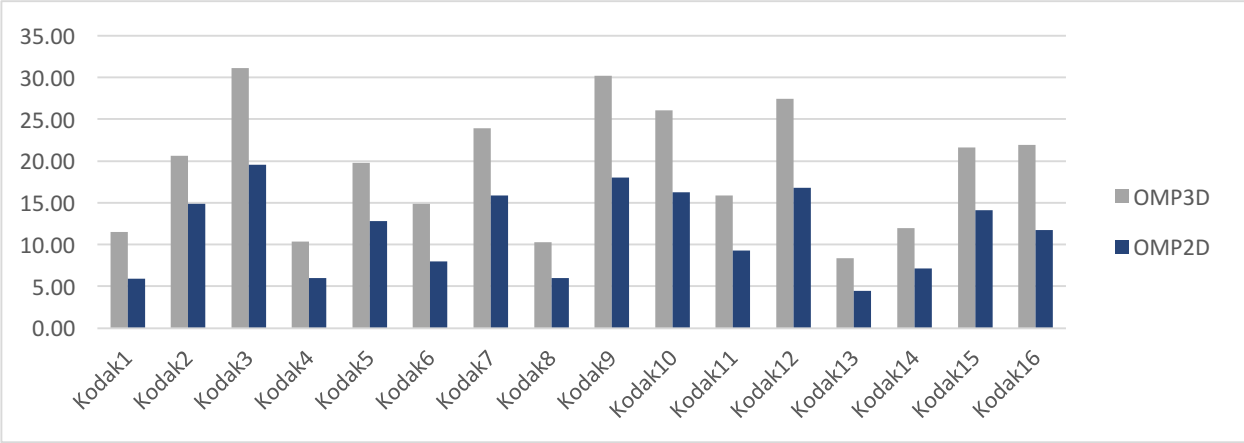
10.2.1.2 Block Size 8

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
OMP3D																		
PSNR	40.94	41.09	41.74	40.86	41.08	41.19	41.69	40.84	41.09	41.09	41.21	41.33	40.81	40.86	41.48	41.29		
Sparsity Ratio	9.84	16.00	23.29	9.26	15.41	12.35	18.58	9.24	22.97	20.19	13.53	20.77	7.36	10.52	17.26	17.27	15.24	4.98
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.92	0.96	0.90	0.99	0.99	0.97	0.98	0.97		
OMP2D																		
PSNR	40.73	41.12	42.03	40.70	41.03	41.16	41.98	40.63	41.22	41.10	41.19	41.41	40.61	40.64	41.78	41.30		
Sparsity Ratio	5.16	11.51	14.53	5.35	10.02	6.72	12.07	5.25	13.47	12.31	7.77	12.55	3.99	6.19	11.14	9.40	9.22	3.32
SSIM	0.92	0.69	0.61	0.95	0.78	0.81	0.6993	0.94	0.52	0.67	0.81	0.66	0.97	0.92	0.64	0.77		
% Increase	90.84	39.00	60.26	73.13	53.81	83.76	53.85	75.98	70.51	63.98	74.13	65.51	84.39	69.96	54.94	83.64	68.61	13.49



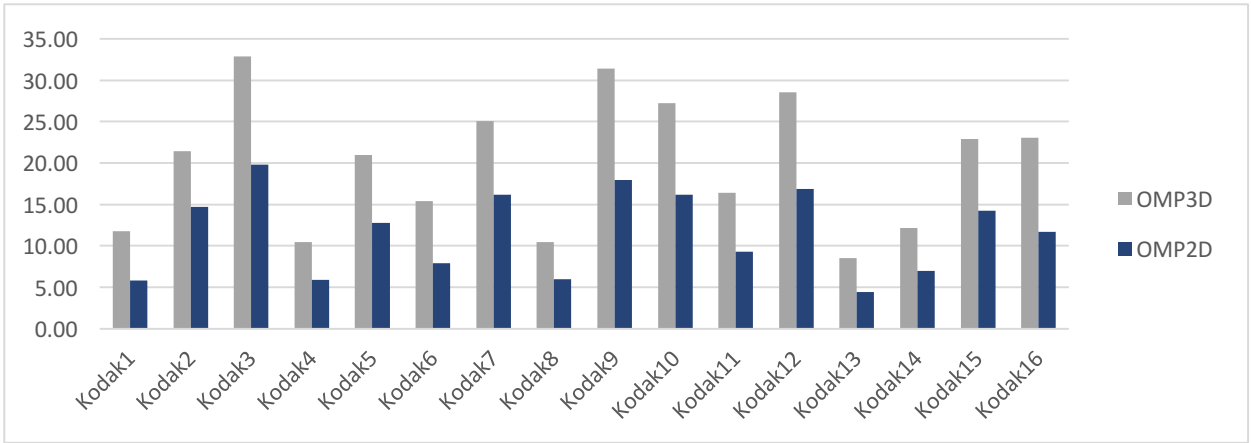
10.2.1.3 Block Size 16

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
OMP3D																		
PSNR	40.62	40.69	41.13	40.59	40.71	40.82	41.11	40.60	40.77	40.71	40.87	40.85	40.58	40.60	41.01	40.84		
Sparsity Ratio	11.50	20.61	31.10	10.34	19.76	14.86	23.92	10.25	30.23	26.05	15.89	27.41	8.32	11.97	21.63	21.93	19.11	7.22
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.92	0.95	0.90	0.99	0.99	0.97	0.97	0.97		
OMP2D																		
PSNR	40.24	40.48	41.11	40.18	40.44	40.61	40.98	40.18	40.66	40.44	40.64	40.66	40.16	40.18	41.02	40.59		
Sparsity Ratio	5.93	14.86	19.57	6.00	12.78	7.98	15.84	5.97	18.01	16.24	9.28	16.82	4.44	7.13	14.08	11.72	11.67	4.84
SSIM	0.92	0.67	0.58	0.95	0.77	0.80	0.68	0.94	0.51	0.65	0.80	0.63	0.97	0.92	0.62	0.75		
% Increase	93.95	38.70	58.93	72.35	54.56	86.12	50.96	71.52	67.90	60.38	71.21	62.99	87.26	67.81	53.61	87.04	67.83	14.75



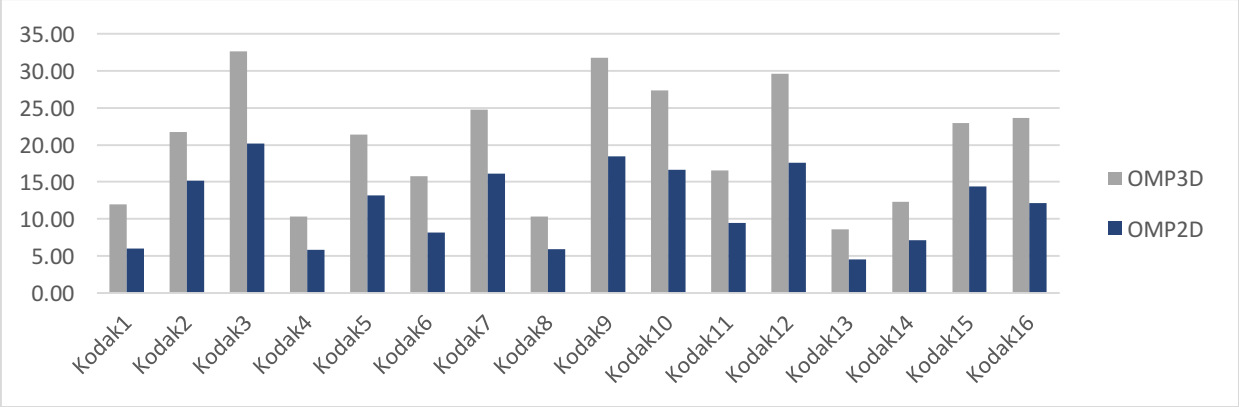
10.2.1.4 Block Size 24

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
OMP3D																		
PSNR	40.55	40.59	40.90	40.54	40.60	40.69	40.90	40.55	40.67	40.60	40.72	40.70	40.54	40.55	40.82	40.70		
Sparsity Ratio	11.74	21.45	32.86	10.45	20.97	15.42	25.11	10.43	31.42	27.25	16.41	28.59	8.54	12.14	22.92	23.09	19.92	7.69
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.91	0.95	0.91	0.99	0.99	0.97	0.97	0.97		
OMP2D																		
PSNR	40.60	40.73	41.19	40.58	40.71	40.93	41.09	40.59	40.87	40.70	40.92	40.85	40.58	40.58	41.12	40.88		
Sparsity Ratio	5.85	14.69	19.80	5.89	12.74	7.92	16.16	5.94	17.93	16.18	9.29	16.88	4.43	6.96	14.20	11.72	11.66	4.90
SSIM	0.93	0.68	0.59	0.95	0.77	0.80	0.69	0.94	0.51	0.66	0.81	0.64	0.97	0.93	0.62	0.76		
% Increase	100.85	46.06	65.99	77.41	64.68	94.65	55.35	75.67	75.26	68.44	76.76	69.36	92.84	74.32	61.35	97.10	74.76	14.88



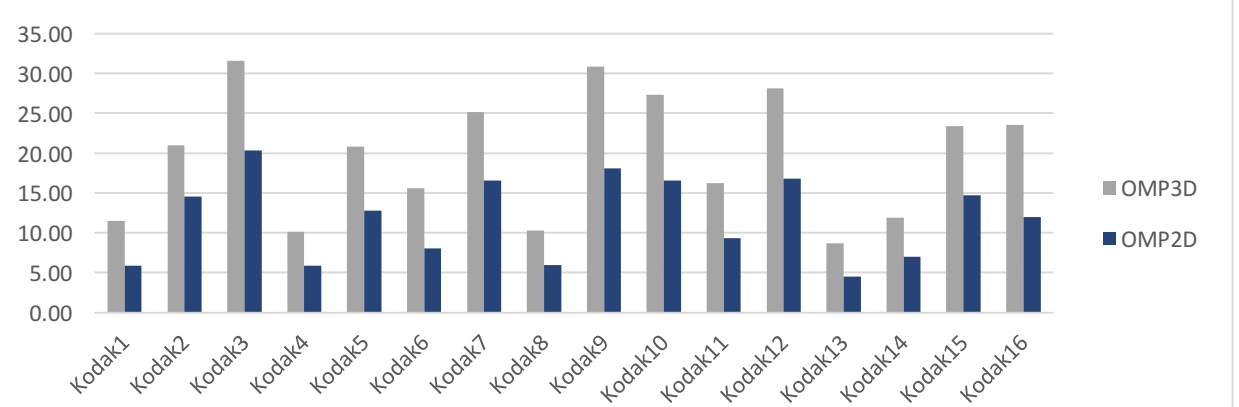
10.2.1.5 Block Size 32

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
OMP3D																		
PSNR	40.53	40.54	40.78	40.52	40.56	40.64	40.75	40.53	40.61	40.56	40.72	40.63	40.52	40.52	40.74	40.63		
Sparsity Ratio	11.93	21.76	32.65	10.32	21.39	15.75	24.74	10.35	31.74	27.39	16.56	29.57	8.62	12.30	22.94	23.61	20.10	7.76
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.91	0.95	0.90	0.99	0.99	0.97	0.97	0.97		
OMP2D																		
PSNR	40.55	40.64	40.99	40.54	40.63	40.84	40.92	40.55	40.75	40.62	40.85	40.76	40.54	40.54	40.94	40.76		
Sparsity Ratio	6.01	15.18	20.14	5.85	13.19	8.15	16.14	5.92	18.45	16.66	9.41	17.55	4.48	7.10	14.39	12.14	11.92	5.05
SSIM	0.93	0.69	0.60	0.96	0.78	0.80	0.69	0.95	0.52	0.67	0.81	0.65	0.97	0.93	0.63	0.76		
% Increase	98.41	43.35	62.15	76.36	62.21	93.37	53.24	74.68	72.00	64.34	76.03	68.49	92.32	73.28	59.48	94.45	72.76	15.21



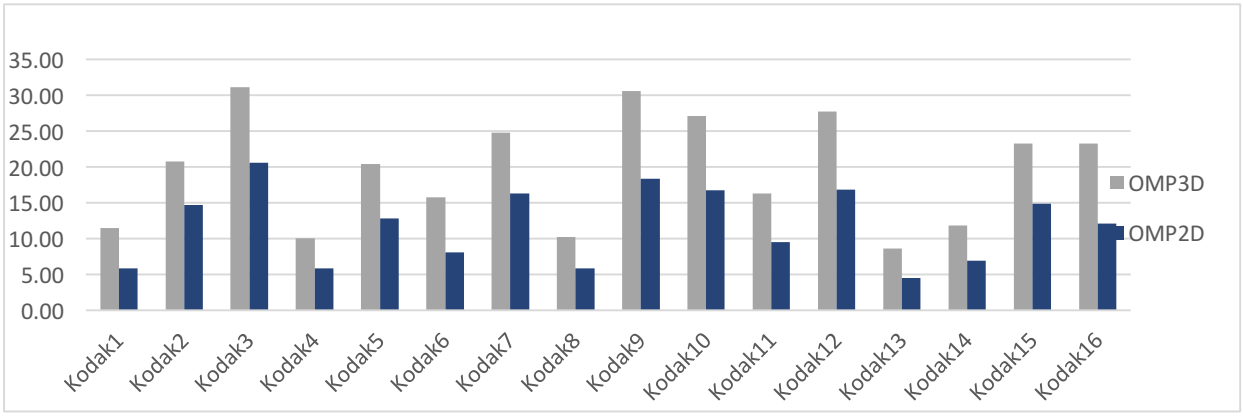
10.2.1.6 Block Size 40

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
OMP3D																		
PSNR	40.52	40.53	40.71	40.51	40.54	40.60	40.69	40.52	40.58	40.53	40.66	40.58	40.51	40.52	40.73	40.60		
Sparsity Ratio	11.54	20.99	31.60	10.16	20.82	15.59	25.19	10.29	30.84	27.32	16.25	28.14	8.67	11.90	23.39	23.56	19.77	7.56
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.98	0.92	0.95	0.91	0.99	0.99	0.98	0.97	0.97		
OMP2D																		
PSNR	40.53	40.58	40.86	40.53	40.59	40.78	40.81	40.53	40.71	40.57	40.76	40.65	40.53	40.53	40.90	40.69		
Sparsity Ratio	5.84	14.58	20.33	5.85	12.83	8.05	16.54	5.94	18.08	16.54	9.35	16.84	4.51	6.97	14.75	12.00	11.81	5.01
SSIM	0.93	0.71	0.59	0.95	0.78	0.79	0.70	0.95	0.51	0.67	0.80	0.66	0.97	0.93	0.61	0.75		
% Increase	97.50	43.92	55.44	73.69	62.30	93.77	52.31	73.31	70.56	65.20	73.77	67.07	92.27	70.69	58.57	96.26	71.66	15.67



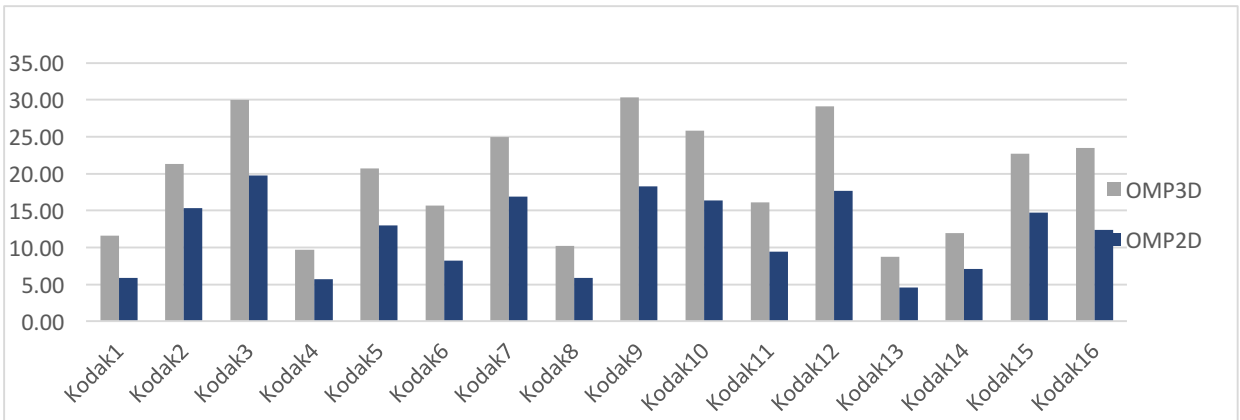
10.2.1.7 Block Size 48

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
OMP3D																		
PSNR	40.51	40.52	40.65	40.51	40.52	40.57	40.61	40.51	40.56	40.52	40.66	40.56	40.51	40.51	40.68	40.56		
Sparsity Ratio	11.50	20.83	31.11	10.06	20.47	15.75	24.77	10.21	30.60	27.12	16.32	27.78	8.68	11.90	23.33	23.31	19.61	7.43
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.92	0.95	0.91	0.99	0.99	0.98	0.97	0.97		
OMP2D																		
PSNR	40.52	40.55	40.75	40.52	40.55	40.72	40.71	40.52	40.65	40.55	40.74	40.61	40.52	40.52	40.91	40.62		
Sparsity Ratio	5.86	14.71	20.61	5.85	12.84	8.12	16.34	5.92	18.37	16.79	9.51	16.85	4.53	6.98	14.86	12.10	11.89	5.07
SSIM	0.93	0.71	0.60	0.95	0.78	0.80	0.70	0.95	0.52	0.67	0.80	0.66	0.97	0.94	0.62	0.76		
% Increase	96.34	41.58	50.95	71.96	59.43	93.99	51.64	72.65	66.59	61.48	71.59	64.85	91.63	70.65	56.97	92.65	69.68	16.13



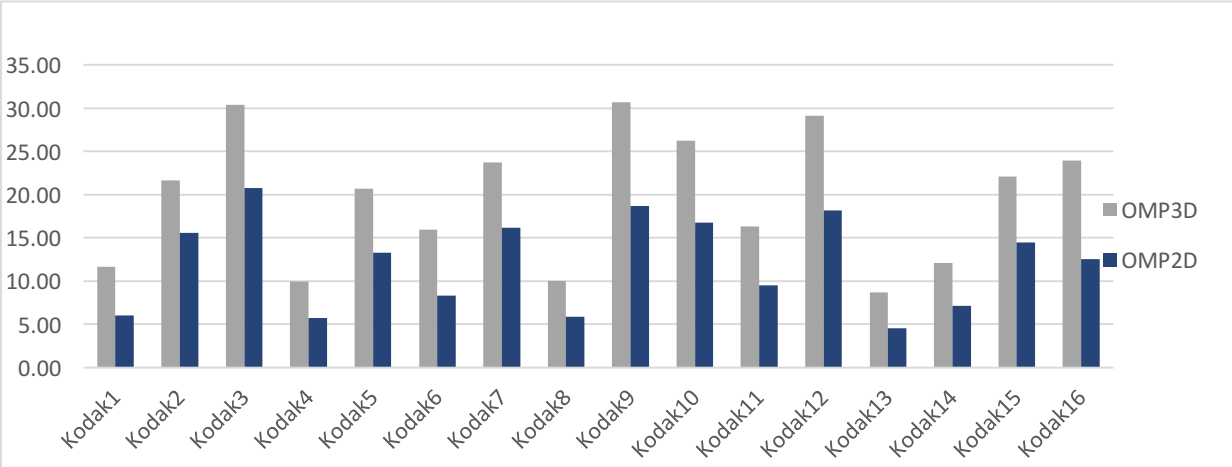
10.2.1.8 Block Size 56

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
OMP3D																		
PSNR	40.51	40.51	40.61	40.51	40.52	40.56	40.58	40.51	40.56	40.51	40.68	40.56	40.51	40.51	40.62	40.55		
Sparsity Ratio	11.59	21.36	29.96	9.72	20.68	15.69	24.92	10.19	30.33	25.86	16.11	29.11	8.70	11.94	22.67	23.47	19.52	7.35
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.98	0.97	0.91	0.95	0.90	0.99	0.99	0.97	0.97	0.97		
OMP2D																		
PSNR	40.51	40.52	40.72	40.51	40.54	40.64	40.67	40.51	40.63	40.54	40.77	40.62	40.51	40.51	40.80	40.61		
Sparsity Ratio	5.91	15.29	19.76	5.65	12.97	8.17	16.89	5.89	18.31	16.37	9.43	17.69	4.54	7.08	14.68	12.35	11.94	5.06
SSIM	0.93	0.70	0.60	0.96	0.79	0.80	0.68	0.95	0.49	0.66	0.81	0.64	0.97	0.93	0.63	0.76		
% Increase	95.99	39.72	51.63	71.99	59.42	91.99	47.62	72.95	65.61	58.00	70.87	64.56	91.76	68.66	54.43	89.97	68.45	16.41



10.2.1.9 Block Size 64

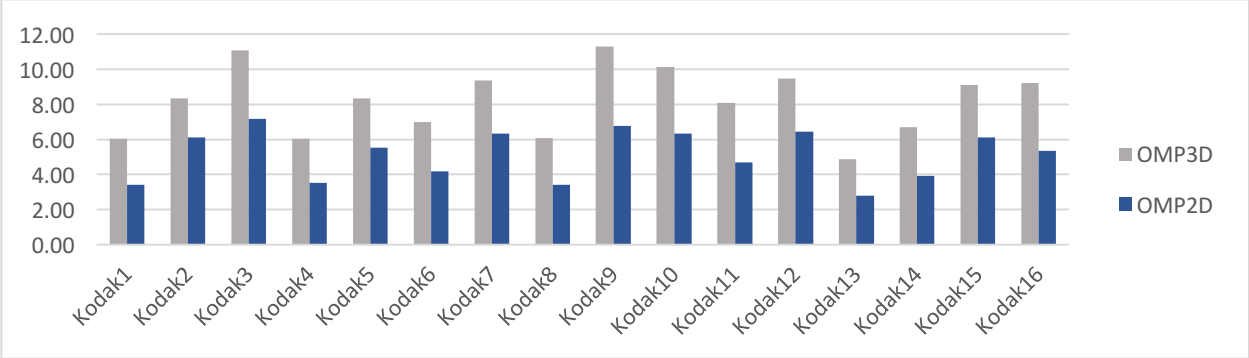
Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
OMP3D																		
PSNR	40.51	40.51	40.59	40.50	40.52	40.54	40.53	40.51	40.54	40.51	40.61	40.54	40.51	40.51	40.62	40.53		
Sparsity Ratio	11.63	21.66	30.41	9.94	20.69	15.92	23.69	10.00	30.72	26.24	16.32	29.15	8.70	12.09	22.09	23.98	19.58	7.38
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.98	0.91	0.95	0.91	0.99	0.99	0.97	0.97	0.97		
OMP2D																		
PSNR	40.51	40.54	40.65	40.51	40.54	40.57	40.62	40.51	40.60	40.53	40.70	40.55	40.51	40.51	40.74	40.57		
Sparsity Ratio	6.04	15.60	20.73	5.73	13.25	8.33	16.15	5.85	18.67	16.76	9.54	18.17	4.54	7.16	14.47	12.55	12.10	5.19
SSIM	0.94	0.70	0.60	0.96	0.78	0.80	0.70	0.95	0.52	0.68	0.81	0.65	0.97	0.93	0.64	0.77		
% Increase	92.55	38.86	46.72	73.53	56.12	91.18	46.67	70.97	64.51	56.56	71.08	60.44	91.45	68.87	52.64	91.11	67.08	16.94



10.2.2 Detailed results for Sparsity Achieved with MP3D and MP2D

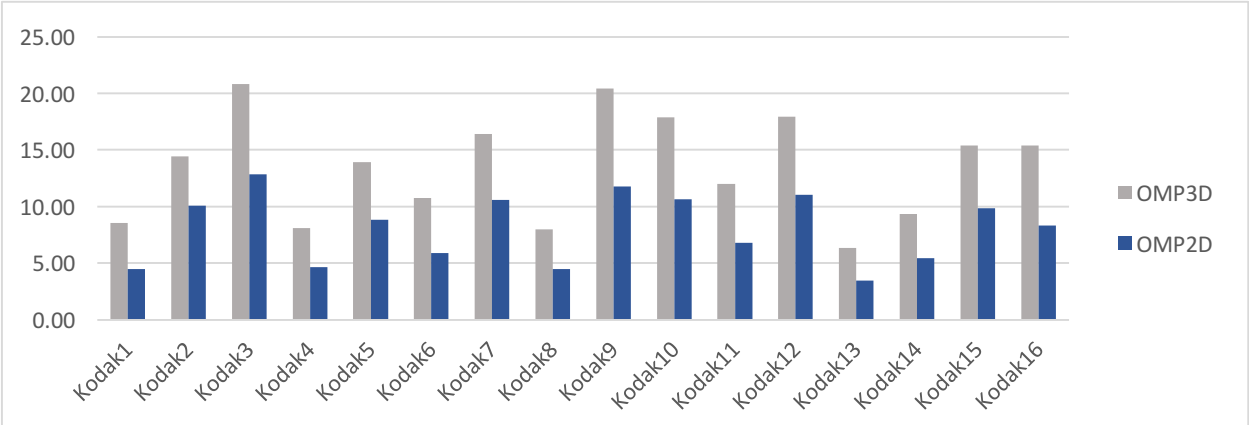
10.2.2.1 Block Size 4

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
MP3D																		
PSNR	41.59	41.91	42.50	41.51	41.82	41.72	42.60	41.45	41.77	41.83	41.90	42.03	41.39	41.53	42.24	42.06		
Sparsity Ratio	6.02	8.35	11.08	6.04	8.34	6.99	9.36	6.06	11.30	10.12	8.10	9.47	4.86	6.70	9.09	9.21	8.19	1.86
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.93	0.96	0.90	0.99	0.99	0.98	0.98	0.98		
MP2D																		
PSNR	42.44	42.87	43.82	42.48	42.73	42.86	43.72	42.33	42.62	42.67	42.91	43.14	42.26	42.37	43.49	43.09		
Sparsity Ratio	3.40	6.12	7.18	3.52	5.53	4.19	6.31	3.41	6.78	6.33	4.68	6.44	2.78	3.92	6.12	5.33	5.13	1.37
SSIM	0.95	0.77	0.69	0.97	0.84	0.85	0.75	0.96	0.62	0.74	0.84	0.74	0.98	0.95	0.71	0.82		
% Increase	76.96	36.49	54.20	71.46	50.63	66.81	48.33	77.78	66.70	59.85	73.06	47.15	74.49	70.74	48.52	72.81	62.25	12.53



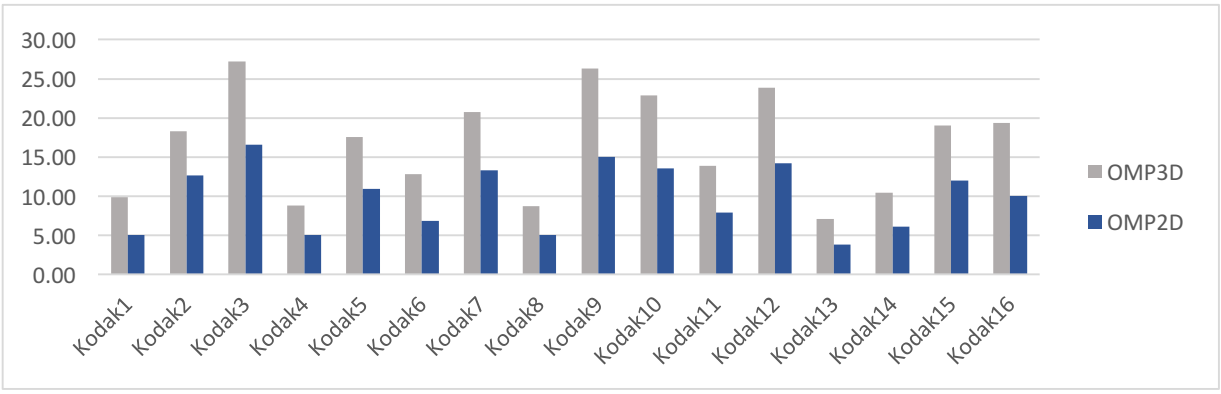
10.2.2.2 Block Size 8

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
MP3D																		
PSNR	40.85	41.03	41.62	40.80	41.03	41.02	41.66	40.78	41.04	41.04	41.14	41.21	40.75	40.81	41.45	41.25		
Sparsity Ratio	8.52	14.44	20.81	8.08	13.92	10.72	16.39	8.00	20.42	17.89	11.99	17.95	6.36	9.33	15.38	15.38	13.47	4.47
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.92	0.96	0.90	0.99	0.99	0.97	0.98	0.97		
MP2D																		
PSNR	41.11	41.47	42.32	41.08	41.41	41.54	42.24	41.03	41.46	41.42	41.57	41.74	40.98	41.05	42.12	41.66		
Sparsity Ratio	4.48	10.07	12.86	4.64	8.84	5.88	10.56	4.49	11.74	10.63	6.79	11.02	3.48	5.43	9.84	8.30	8.07	2.93
SSIM	0.93	0.71	0.62	0.96	0.79	0.82	0.70	0.95	0.54	0.69	0.82	0.68	0.97	0.93	0.65	0.78		
% Increase	90.10	43.37	61.74	73.94	57.37	82.33	55.22	78.11	73.91	68.30	76.45	62.92	83.08	71.70	56.37	85.32	70.01	12.49



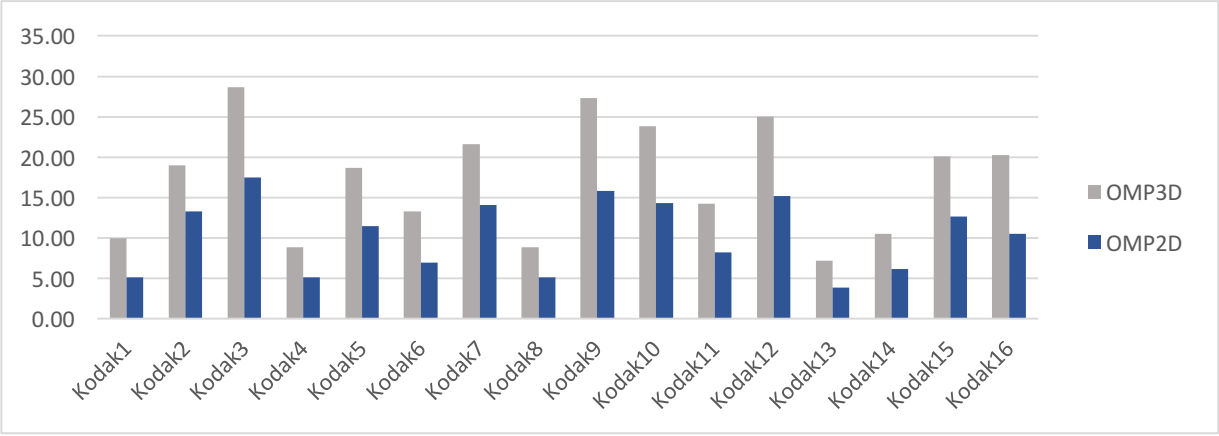
10.2.2.3 Block Size 16

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
MP3D																		
PSNR	40.60	40.68	41.10	40.58	40.69	40.75	41.12	40.59	40.76	40.70	40.84	40.80	40.56	40.58	40.99	40.84		
Sparsity Ratio	9.83	18.29	27.25	8.81	17.57	12.84	20.75	8.73	26.32	22.87	13.86	23.87	7.05	10.41	19.05	19.34	16.68	6.39
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.92	0.95	0.90	0.99	0.99	0.97	0.97	0.97		
MP2D																		
PSNR	40.69	40.90	41.48	40.65	40.86	41.02	41.37	40.65	41.00	40.86	41.05	41.06	40.63	40.65	41.37	41.01		
Sparsity Ratio	5.05	12.67	16.58	5.06	10.94	6.80	13.29	5.01	15.04	13.55	7.91	14.21	3.78	6.09	11.98	10.05	9.88	4.06
SSIM	0.92	0.69	0.60	0.96	0.78	0.80	0.69	0.95	0.52	0.67	0.81	0.65	0.97	0.93	0.63	0.76		
% Increase	94.62	44.39	64.32	74.24	60.70	88.78	56.06	74.05	74.98	68.81	75.15	68.03	86.78	70.88	58.96	92.42	72.07	13.36



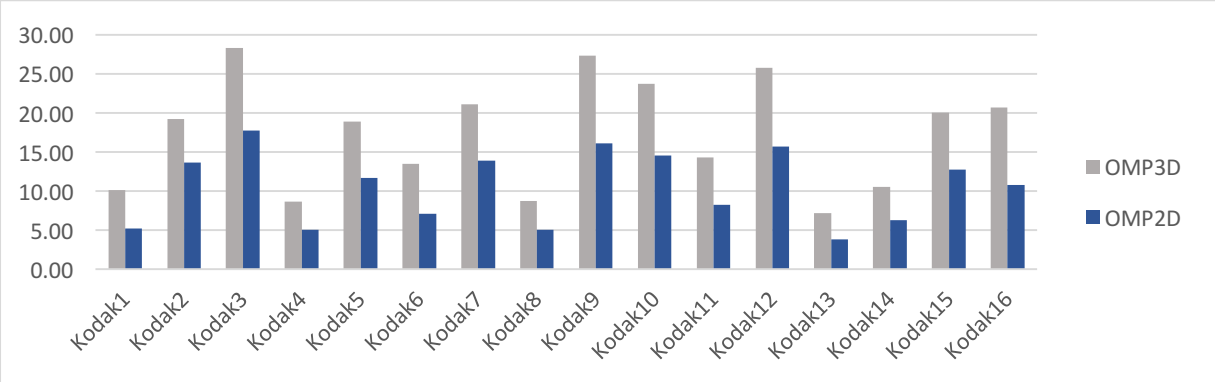
10.2.2.4 Block Size 24

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
MP3D																		
PSNR	40.54	40.58	40.89	40.53	40.59	40.66	40.89	40.54	40.65	40.59	40.67	40.66	40.53	40.54	40.82	40.69		
Sparsity Ratio	9.98	18.97	28.68	8.84	18.63	13.26	21.61	8.83	27.31	23.81	14.25	25.03	7.20	10.49	20.09	20.27	17.33	6.81
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.91	0.95	0.91	0.99	0.99	0.97	0.97	0.97		
MP2D																		
PSNR	40.58	40.72	41.14	40.56	40.69	40.86	41.04	40.57	40.85	40.69	40.86	40.82	40.56	40.57	41.10	40.85		
Sparsity Ratio	5.12	13.31	17.50	5.10	11.45	6.97	14.06	5.14	15.80	14.28	8.21	15.18	3.84	6.16	12.62	10.51	10.33	4.38
SSIM	0.93	0.69	0.59	0.95	0.78	0.80	0.69	0.94	0.51	0.67	0.81	0.64	0.97	0.93	0.62	0.76		
% Increase	94.72	42.56	63.93	73.47	62.75	90.30	53.73	71.79	72.79	66.71	73.61	64.87	87.58	70.22	59.26	92.86	71.32	13.97



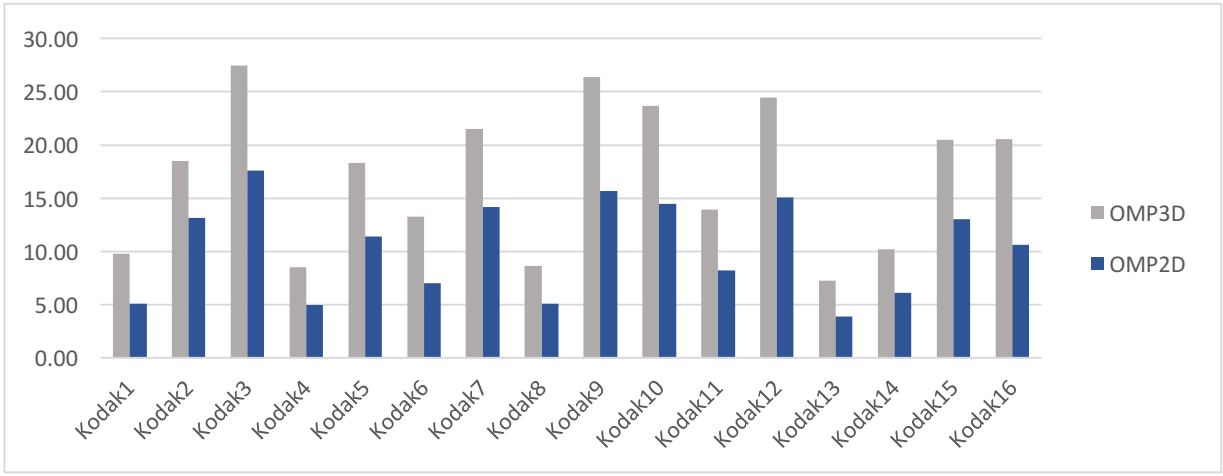
10.2.2.5 Block Size 32

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
MP3D																		
PSNR	40.52	40.54	40.76	40.52	40.55	40.62	40.75	40.52	40.60	40.55	40.70	40.60	40.52	40.52	40.72	40.61		
Sparsity Ratio	10.10	19.21	28.38	8.66	18.89	13.52	21.10	8.71	27.37	23.77	14.30	25.82	7.23	10.59	20.10	20.68	17.40	6.84
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.91	0.95	0.91	0.99	0.99	0.97	0.97	0.97		
MP2D																		
PSNR	40.54	40.61	40.95	40.54	40.61	40.80	40.87	40.54	40.73	40.61	40.78	40.73	40.53	40.54	40.91	40.73		
Sparsity Ratio	5.24	13.68	17.73	5.03	11.73	7.12	13.91	5.10	16.16	14.58	8.29	15.69	3.86	6.26	12.74	10.80	10.49	4.48
SSIM	0.93	0.69	0.60	0.96	0.78	0.80	0.70	0.95	0.52	0.67	0.81	0.65	0.97	0.93	0.63	0.76		
% Increase	92.60	40.37	60.04	72.36	61.02	89.97	51.64	70.90	69.40	63.06	72.61	64.64	87.31	69.15	57.79	91.52	69.65	14.37



10.2.2.6 Block Size 40

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
MP3D																		
PSNR	40.51	40.53	40.70	40.51	40.53	40.58	40.69	40.51	40.58	40.53	40.57	40.56	40.51	40.51	40.72	40.58		
Sparsity Ratio	9.76	18.51	27.49	8.53	18.34	13.26	21.51	8.64	26.39	23.70	13.96	24.47	7.25	10.20	20.46	20.54	17.06	6.64
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.98	0.92	0.95	0.91	0.99	0.99	0.98	0.97	0.97		
MP2D																		
PSNR	40.53	40.58	40.82	40.52	40.58	40.76	40.77	40.53	40.68	40.56	40.61	40.64	40.52	40.52	40.87	40.67		
Sparsity Ratio	5.09	13.14	17.63	4.99	11.40	7.01	14.18	5.09	15.66	14.45	8.22	15.07	3.87	6.10	13.02	10.65	10.35	4.41
SSIM	0.93	0.71	0.59	0.95	0.78	0.79	0.70	0.95	0.51	0.67	0.80	0.66	0.97	0.94	0.61	0.76		
% Increase	91.84	40.82	55.93	70.87	60.92	89.08	51.66	69.97	68.56	64.04	69.71	62.39	87.65	67.23	57.18	92.81	68.79	14.55



10.2.3 Detailed results for performance of OMP3D C++ MEX Implementation

10.2.3.1 Block Size 4

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
PSNR	41.82	42.06	42.73	41.68	41.93	42.05	42.65	41.62	41.92	41.93	42.06	42.29	41.59	41.71	42.35	42.20	42.04	0.33
Sparsity Ratio	6.80	9.01	12.10	6.68	9.06	7.96	10.27	6.75	12.40	11.29	8.89	11.12	5.44	7.36	10.09	10.21	9.09	2.04
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.93	0.96	0.90	0.99	0.99	0.98	0.98	0.98	0.98	0.02
OMP3D	**MATLAB**																	
Run 1	130.7	98.4	73.3	133.7	97.4	114.1	87.8	134.1	71.1	77.7	99.6	79.1	165.6	120.1	87.4	86.4	103.5	26.1
Run 2	130.9	98.1	73.5	133.6	97.5	114.7	87.6	132.7	71.0	78.0	99.5	79.1	164.3	119.9	87.2	86.1	103.4	25.9
Run 3	131.0	98.6	73.3	133.5	98.9	113.1	85.7	132.2	71.4	78.1	99.8	79.2	163.0	120.3	87.2	86.3	103.2	25.6
Average	130.8	98.4	73.4	133.6	97.9	114.0	87.0	133.0	71.2	77.9	99.6	79.1	164.3	120.1	87.3	86.3	103.4	25.9
OMP3D	**MEX**																	
Run 1	12.7	9.9	7.7	13.1	9.8	11.2	8.9	12.9	7.5	8.2	10.1	8.2	15.8	12.0	9.0	8.9	10.4	2.3
Run 2	12.7	10.0	7.7	13.0	9.8	11.2	9.1	14.4	7.6	8.2	10.1	8.3	15.9	11.9	9.0	8.9	10.5	2.4
Run 3	12.8	9.9	7.7	12.9	10.0	11.2	8.8	13.0	7.6	8.2	10.1	8.2	15.6	12.2	9.1	8.9	10.4	2.3
Average	12.7	9.9	7.7	13.0	9.9	11.2	8.9	13.4	7.6	8.2	10.1	8.2	15.8	12.0	9.0	8.9	10.4	2.3
% Increase	90.26	89.92	89.50	90.27	89.93	90.16	89.76	89.90	89.37	89.52	89.87	89.60	90.39	89.99	89.68	89.68	89.86	0.29

10.2.3.2 Block Size 8

Image	Kodak1	Kodak2	Kodak3	Kodak4	Kodak5	Kodak6	Kodak7	Kodak8	Kodak9	Kodak10	Kodak11	Kodak12	Kodak13	Kodak14	Kodak15	Kodak16	Mean	Standard Deviation
PSNR	40.94	41.09	41.74	40.86	41.08	41.19	41.69	40.84	41.09	41.09	41.21	41.33	40.81	40.86	41.48	41.29	41.16	0.28
Sparsity Ratio	9.84	16.00	23.29	9.26	15.41	12.35	18.58	9.24	22.97	20.19	13.53	20.77	7.36	10.52	17.26	17.26	15.24	4.98
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.92	0.96	0.90	0.99	0.99	0.97	0.98	0.97	0.97	0.03
OMP3D	**MATLAB**																	
Run 1	148.6	90.1	63.6	164.1	94.5	121.1	78.8	161.8	62.4	73.0	109.2	69.8	209.3	148.0	84.1	82.2	110.0	42.6
Run 2	149.6	91.1	63.5	161.9	93.7	117.7	77.8	160.4	62.9	72.0	109.1	69.7	209.3	146.8	84.5	82.1	109.5	42.3
Run 3	149.5	90.5	62.3	161.9	94.2	120.3	77.6	162.9	63.6	72.2	109.3	68.7	211.1	144.3	84.2	83.8	109.8	42.7
Average	149.2	90.6	63.1	162.6	94.1	119.7	78.1	161.7	62.9	72.4	109.2	69.4	209.9	146.4	84.3	82.7	109.8	42.5
OMP3D	**MEX**																	
Run 1	33.1	17.8	19.4	32.1	29.3	25.9	15.9	31.6	12.7	14.9	23.6	14.1	48.7	39.8	16.4	16.8	24.5	10.1
Run 2	39.5	17.9	19.5	33.5	29.2	27.6	19.1	39.1	14.6	22.4	25.6	14.0	40.9	43.4	17.8	18.2	26.4	9.7
Run 3	31.5	18.0	19.3	34.3	19.1	33.5	17.1	31.6	13.5	17.4	30.9	18.9	62.9	31.7	19.6	22.2	26.3	11.7
Average	34.7	17.9	19.4	33.3	25.9	29.0	17.4	34.1	13.6	18.2	26.7	15.7	50.8	38.3	17.9	19.1	25.7	10.0
% Increase	76.74	80.21	69.26	79.51	72.52	75.79	77.73	78.91	78.39	74.82	75.56	77.44	75.79	73.84	78.75	76.93	76.39	2.74

10.2.4 Detailed results for performance of SPMP3D C++ MEX Implementation

10.2.4.1 Block Size 4

Image	**Kodak1**	**Kodak2**	**Kodak3**	**Kodak4**	**Kodak5**	**Kodak6**	**Kodak7**	**Kodak8**	**Kodak9**	**Kodak10**	**Kodak11**	**Kodak12**	**Kodak13**	**Kodak14**	**Kodak15**	**Kodak16**	**Mean**	**S.D.**
PSNR	41.59	41.91	42.50	41.51	41.82	41.72	42.60	41.45	41.77	41.83	41.90	42.03	41.39	41.53	42.24	42.06	41.87	0.34
Sparsity Ratio	6.02	8.35	11.08	6.04	8.34	6.99	9.36	6.06	11.30	10.12	8.10	9.47	4.86	6.70	9.09	9.21	8.19	1.86
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.93	0.96	0.90	0.99	0.99	0.98	0.98	0.98	0.98	0.02
MATLAB	145.35	104.83	81.04	148.07	107.91	128.82	96.39	147.52	79.04	86.19	110.02	96.85	183.26	134.08	96.73	95.81	115.12	28.63
C++ MEX	13.39	10.36	8.42	13.48	10.44	12.03	9.55	13.48	8.40	8.85	10.65	9.88	16.15	12.39	9.64	9.65	11.05	2.14
% Increase	90.78	90.12	89.62	90.89	90.32	90.66	90.09	90.87	89.38	89.73	90.32	89.80	91.19	90.76	90.03	89.93	90.28	0.51

10.2.4.2 Block Size 8

Image	**Kodak1**	**Kodak2**	**Kodak3**	**Kodak4**	**Kodak5**	**Kodak6**	**Kodak7**	**Kodak8**	**Kodak9**	**Kodak10**	**Kodak11**	**Kodak12**	**Kodak13**	**Kodak14**	**Kodak15**	**Kodak16**	**Mean**	**S.D.**
PSNR	40.85	41.03	41.62	40.80	41.03	41.02	41.66	40.78	41.04	41.04	41.14	41.21	40.75	40.81	41.45	41.25	41.09	0.28
Sparsity Ratio	8.52	14.44	20.81	8.08	13.92	10.72	16.39	8.00	20.42	17.89	11.99	17.95	6.36	9.33	15.38	15.38	13.47	4.47
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.92	0.96	0.90	0.99	0.99	0.97	0.98	0.97	0.97	0.03
MATLAB	177.88	104.34	72.23	184.38	110.67	140.08	91.67	185.80	73.34	83.93	127.45	86.35	230.50	159.68	96.16	97.36	126.36	46.45
C++ MEX	23.37	14.16	9.78	24.13	14.28	18.98	12.45	24.38	10.12	11.33	16.94	11.66	30.68	21.17	13.10	13.09	16.85	6.02
% Increase	86.86	86.43	86.46	86.91	87.10	86.45	86.42	86.88	86.21	86.50	86.71	86.50	86.69	86.74	86.38	86.56	86.61	0.23

10.2.4.3 Block Size 16

Image	**Kodak1**	**Kodak2**	**Kodak3**	**Kodak4**	**Kodak5**	**Kodak6**	**Kodak7**	**Kodak8**	**Kodak9**	**Kodak10**	**Kodak11**	**Kodak12**	**Kodak13**	**Kodak14**	**Kodak15**	**Kodak16**	**Mean**	**S.D.**
PSNR	40.60	40.68	41.10	40.58	40.69	40.75	41.12	40.59	40.76	40.70	40.84	40.80	40.56	40.58	40.99	40.84	40.76	0.17
Sparsity Ratio	9.83	18.29	27.25	8.81	17.57	12.84	20.75	8.73	26.32	22.87	13.86	23.87	7.05	10.41	19.05	19.34	16.68	6.39
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.92	0.95	0.90	0.99	0.99	0.97	0.97	0.97	0.97	0.03
MATLAB	492.23	263.04	178.51	544.95	274.04	375.81	231.92	548.67	181.58	208.97	349.33	203.59	686.94	462.47	251.59	248.61	343.89	152.13
C++ MEX	120.06	64.04	43.54	134.76	66.94	91.65	56.75	135.28	44.73	52.74	85.65	49.93	167.86	112.85	62.07	60.73	84.35	37.25
% Increase	75.61	75.65	75.61	75.27	75.57	75.61	75.53	75.34	75.36	74.76	75.48	75.47	75.56	75.60	75.33	75.57	75.46	0.21

10.2.4.4 Block Size 24

Image	**Kodak1**	**Kodak2**	**Kodak3**	**Kodak4**	**Kodak5**	**Kodak6**	**Kodak7**	**Kodak8**	**Kodak9**	**Kodak10**	**Kodak11**	**Kodak12**	**Kodak13**	**Kodak14**	**Kodak15**	**Kodak16**	**Mean**	**S.D.**
PSNR	40.54	40.58	40.89	40.53	40.59	40.66	40.89	40.54	40.65	40.59	40.67	40.66	40.53	40.54	40.82	40.69	40.65	0.12
Sparsity Ratio	9.98	18.97	28.68	8.84	18.63	13.26	21.61	8.83	27.31	23.81	14.25	25.03	7.20	10.49	20.09	20.27	17.33	6.81
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.91	0.95	0.91	0.99	0.99	0.97	0.97	0.97	0.97	0.03
MATLAB	858.65	455.51	298.32	970.20	460.30	643.25	393.62	968.86	312.51	359.68	602.42	344.07	1189.08	814.38	427.94	421.87	595.04	271.07
C++ MEX	294.31	154.70	102.40	332.74	158.44	221.43	136.39	333.41	107.84	121.93	206.56	117.75	407.13	280.00	145.94	144.80	204.11	93.05
% Increase	65.72	66.04	65.67	65.70	65.58	65.58	65.35	65.59	65.49	66.10	65.71	65.78	65.76	65.62	65.90	65.68	65.70	0.18

10.2.4.5 Block Size 32

Image	**Kodak1**	**Kodak2**	**Kodak3**	**Kodak4**	**Kodak5**	**Kodak6**	**Kodak7**	**Kodak8**	**Kodak9**	**Kodak10**	**Kodak11**	**Kodak12**	**Kodak13**	**Kodak14**	**Kodak15**	**Kodak16**	**Mean**	**S.D.**
PSNR	40.52	40.54	40.76	40.52	40.55	40.62	40.75	40.52	40.60	40.55	40.70	40.60	40.52	40.52	40.72	40.61	40.60	0.09
Sparsity Ratio	10.10	19.21	28.38	8.66	18.89	13.52	21.10	8.71	27.37	23.77	14.30	25.82	7.23	10.59	20.10	20.68	17.40	6.84
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.97	0.91	0.95	0.91	0.99	0.99	0.97	0.97	0.97	0.97	0.03
MATLAB	1294.92	680.37	461.65	1510.70	691.34	961.94	618.37	1498.33	477.70	549.31	912.68	508.21	1808.11	1232.48	651.83	632.93	905.68	416.58
C++ MEX	450.07	236.06	160.27	523.50	240.15	336.91	216.17	520.15	166.22	191.53	318.13	176.37	628.64	428.55	227.11	220.99	315.05	144.54
% Increase	65.24	65.30	65.28	65.35	65.26	64.98	65.04	65.28	65.20	65.13	65.14	65.30	65.23	65.23	65.16	65.08	65.20	0.10

10.2.4.6 Block Size 40

Image	**Kodak1**	**Kodak2**	**Kodak3**	**Kodak4**	**Kodak5**	**Kodak6**	**Kodak7**	**Kodak8**	**Kodak9**	**Kodak10**	**Kodak11**	**Kodak12**	**Kodak13**	**Kodak14**	**Kodak15**	**Kodak16**	**Mean**	**S.D.**
PSNR	40.51	40.53	40.70	40.51	40.53	40.58	40.69	40.51	40.58	40.53	40.57	40.56	40.51	40.51	40.72	40.58	40.57	0.07
Sparsity Ratio	9.76	18.51	27.49	8.53	18.34	13.26	21.51	8.64	26.39	23.70	13.96	24.47	7.25	10.20	20.46	20.54	17.06	6.64
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.98	0.92	0.95	0.91	0.99	0.99	0.98	0.97	0.97	0.97	0.03
MATLAB	1847.35	975.34	655.10	2119.93	987.19	1367.53	841.43	2096.83	682.32	763.71	1297.42	740.19	2490.58	1770.42	883.69	775.56	1268.41	586.01
C++ MEX	676.09	356.59	239.71	773.81	360.07	500.04	307.01	762.47	252.37	279.15	474.69	270.54	907.31	647.24	323.01	264.37	462.15	214.32
% Increase	63.40	63.44	63.41	63.50	63.53	63.43	63.51	63.64	63.01	63.45	63.41	63.45	63.57	63.44	63.45	65.91	63.60	0.61

10.2.4.7 Block Size 48

Image	**Kodak1**	**Kodak2**	**Kodak3**	**Kodak4**	**Kodak5**	**Kodak6**	**Kodak7**	**Kodak8**	**Kodak9**	**Kodak10**	**Kodak11**	**Kodak12**	**Kodak13**	**Kodak14**	**Kodak15**	**Kodak16**	**Mean**	**S.D.**
PSNR	40.51	40.52	40.63	40.51	40.52	40.55	40.61	40.51	40.55	40.52	40.55	40.54	40.51	40.51	40.63	40.55	40.54	0.04
Sparsity Ratio	9.68	18.34	26.88	8.41	17.97	13.42	21.11	8.56	26.22	23.37	13.96	24.24	7.23	10.18	20.33	20.34	16.89	6.52
SSIM	0.99	1.00	0.98	0.99	0.99	0.99	0.99	0.98	0.92	0.95	0.91	0.99	0.99	0.98	0.97	0.97	0.97	0.03
MATLAB	2453.70	1264.83	774.65	2818.96	1310.25	1758.21	1061.30	2700.25	900.18	990.03	1651.65	961.93	3220.43	2293.44	1146.56	1127.57	1652.12	765.73
C++ MEX	785.16	482.56	314.18	1060.67	489.18	655.93	280.43	1000.07	335.93	253.15	629.61	353.23	1183.88	840.10	421.00	420.28	594.08	288.72
% Increase	68.00	61.85	59.44	62.37	62.66	62.69	73.58	62.96	62.68	74.43	61.88	63.28	63.24	63.37	63.28	62.73	64.28	4.01

10.3 Appendix 3 - Test Images

In this appendix, we present the images that the evaluation was conducted on. The images are displayed here sorted by their Maximum Sparsity Ratio Achieved (MSRA) using OMP3D, most significant gain in sparsity to least significant. These standard images are often used for comparison of compression and image processing techniques and can be acquired from (Kodak, 2010).

Figure 10-1 - 'Kodak3.png' - MSRA 32.863

Figure 10-2 - 'Kodak9.png' - MSRA 31.739

Figure 10-3 - 'Kodak12.png' - MSRA 29.565

Figure 10-4 - 'Kodak10.png' - MSRA 27.386

Figure 10-5 - 'Kodak7.png' - MSRA 25.192

Figure 10-6 - 'Kodak16.png' - MSRA 23.979

Figure 10-7 - 'Kodak15.png' - MSRA 23.391

Figure 10-8 - 'Kodak2.png' - MSRA 21.656

Figure 10-9 - 'Kodak5.png' - MSRA 21.393

Figure 10-10 - 'Kodak11.png' - MSRA 16.558

Figure 10-11 - 'Kodak6.png' - MSRA

Figure 10-12 - 'Kodak14.png' - MSRA 12.304

Figure 10-13 - 'Kodak1.png' - MSRA 11.929

Figure 10-14 - 'Kodak8.png' - MSRA 13.429

Figure 10-15- 'Kodak4.png' - MSRA 10.448

Figure 10-16 - 'Kodak13.png' - MSRA 8.697

10.4 Appendix 4 - Project Diary

Date	Description
26/07/2016	Project Proposal was accepted and added to my list of choices.
04/10/2016	First meeting with Dr Vogiatzis discussed the general idea of the project and the possibilities that could be explored e.g. applications to compression and use of GPU for acceleration. Also discussed a previous project which did an implementation in the 2D space using Java, and potentially carrying on using Java in the same fashion. This was advised against, due to it being slow at number crunching and fairly syntax heavy as framework. Python using NumPy as a framework was suggested.
07/10/2016	Met with Dr Rebollo-Neira, to discuss high-level the concept of the 3D version of the algorithm and how it differs from the 2D version.
11/10/2016	Meeting with Dr Vogiatzis, we discussed the possibilities of using Python rather than Java or over other languages. It was discussed that as part of Dr Rebollo-Neira's research, various C++ source code already exists, some of which could be used to prevent the duplication of effort. This then shifted the implementation towards C++/C instead of Python. This also offers performance benefits since being compiled. I was also advised to see if there was any other research of this algorithm that already exists.
18/10/2016	Reported back stating I couldn't find anything with the same idea as this project - i.e. 3D objects, this algorithm, C++.
20/10/2016	Met with Dr Rebollo-Neira and went through the technicalities of the SPMP2D algorithm (so that the same idea can be applied to the 3D version). This included getting the MATLAB version to compile since I originally had some issues (due to a few missing functions).
21/10/2016	Submitted Project Definition Form outlining the scope of the project.
24/10/2016	Met with Dr Vogiatzis and explained I was trying to understand the algorithm on a more detailed level, further discussed the definition of the project and the scope.
26/10/2016	Met with Dr Rebollo-Neira to discuss the SPMP3D algorithm implementation. Initial results from MATLAB show that it actually offers higher sparsity than doing the 2D version multiple times for each dimension. For now, we are just applying the routine on colour images, with each colour channel being a different dimension in the 3D space. Task was to try and understand the algorithm in more detail to begin coding something the next week.
04/11/2016	Met with Dr Rebollo-Neira, we discussed the creation of MEX files for IPSP3D and IP3D functions since it was discovered from profiling these functions are the bottlenecks of the entire algorithm. This meant looking briefly at how Mex files are compiled and called from MATLAB.

08/11/2016	Met with Dr Vogiatzis and explained some of the issues I was facing with C++, i.e. problems debugging and memory leaks causing the program to crash. Offered advice about tools/profilers that can be used to debug C++ programs.
11/11/2016	In this meeting with Dr Rebollo-Neira, I demonstrated the IPSP3D function working as pure C++ and an attempt at the MEX version so that is callable from MATLAB, however memory leaks existed which made it very unstable, task for the next week was to fix this. Also discussed the GPU desktop that was acquired a few years ago to look at its GPU prospectives / specs.
22/11/2016	Fixed the memory leaks in the IPSP3D function so that is now callable from MATLAB, run time for this function in the entire algorithm is roughly ~14 times quicker that the MATLAB implementation, this has a major effect on the efficiency of the entire algorithm.
25/11/2016	Demonstrated the speed improvements alone from IPSP3D to Dr Vogiatzis, and outlined the plan in terms of doing the same on other 'bottlenecking' functions.
26/11/2016	Met with Dr Rebollo-Neira, and fixed some compilation issues when running it on different machines. We then discussed for comparison purposes turning some 2D functions into MEX and comparing the speed efficiency against the already implemented C++ routine. This would determine whether it is justifiable to do the entire SPMP3D routine as C++ or whether there is little to gain.
29/11/2016	Discussed the GPU prospects in more detail in terms of memory allowance and some operations which could be trickier on GPU. e.g. finding the maximum value in an array which is naturally quicker on CPU since the algorithm needs to be changed for it to work on the GPU. Advised it is possible to do these operations on the GPU.
01/12/2016	Completed the IP3D function where speed increase is approximately ~3x on the entire algorithm.
08/12/2016	Looked at a journal about implementing OMP (Orthogonal Matching Pursuit) on the GPU. Since OMP is more greedy in terms of memory, SPMP should be fine on the GPU. Also via email discussed another bottlenecking function *'hnew'* which would be completed for the following week.
10/12/2016	Meeting with Dr Rebollo-Neira- Beginning to convert a few functions in the OMP routine to C++ so that comparison between OMP3D and SPMP3D can be made. These routines (Since they are bottlenecks) are Orthogonalize, Reorthohonalize and Biorthogonalize. The other functions made so far are also useable in this algorithm. Also discussed that it is worthwhile starting the report, at least the headings, bullet points etc.
13/12/2016	Met with Dr Vogiatzis to discuss plans over Christmas Break, explained main focus would be on exams and agreed that making a start on the final report is a good idea.
26/01/2017	Large gap due to revision for exam period. Creation of the OMP3D routine in C++ so that it can be called from MATLAB. Meeting with Dr Rebollo-Neira to reflect on the algorithm

10/02/2017	Discussion on the storage requirements of the GPU application for the SPMP3D routine was discussed with Dr Rebollo-Neira. Went into more detail about utilisation of fast-access memory and the storage limitations. It was determined that the storage of the dictionaries and small block size of 8 x 8 x 3 will be problem free.
20/02/2017	Moved into the PhD. room provided by the department of mathematics so that I can use the more advanced GPU hardware for programming. CUDA programming begun at this point.
13/03/2017	Further optimisations made with the OMP3D and SPMP3D routine by making more effective use of the BLAS calls. This indicates another significant improvement in performance when samples tests were run with Dr Rebollo-Neira.
27/03/2017	First draft of the SPMP3D routine is working one block. This takes approximately 63ms to complete. Extrapolating this to a whole image of size 256 x 256, then a 1.5s execution time is expected.
07/04/2017	Begun numerical tests for comparisons between approximating channels of colour images simultaneously and individually through the use of the 3D and 2D routines respectively. Discussed the preliminary results with Dr Rebollo-Neira
24/04/2017	Met with Dr Vogiatzis to discuss a problem encountered on the GPU when scaling the number of blocks up. Offered advice by scaling back the complexity until the problem was found. Also discussed structure of the demo/viva.
28/04/2017	Final report drafted, meeting with Dr Rebollo-Neira to review the report and conclusions. MP3D was successful completed on the GPU so that the sparsity ratio can be determined. Initial results show it to be similar, but ever so slightly better than the C++ routines.

10.5 Appendix 5 - List of Tools Used

Below is a list of tools and programs that were used for the duration of this project:

- **AMD CodeAnalyst for Windows**– Tool that was used for profiling C++ function calls: http://developer.amd.com/tools-and-sdks/archive/compute/amd-codeanalyst-performance-analyzer/codeanalyst-performance-analyzer-for-windows/
- **CLion** – IDE that was used for development of the C++ MEX files / CUDA files. https://www.jetbrains.com/clion/
- **CUDA development toolkit V8.0**- For CUDA runtime, libraries and NVCC compiler . https://developer.nvidia.com/cuda-toolkit
- **cuda-memcheck** – Tool that was used for checking memory violations on CUDA applications. Distributed with the CUDA development toolkit. https://developer.nvidia.com/cuda-toolkit
- **cuda-gdb** – Tool that was used for debugging CUDA applications on Linux. Distributed with the CUDA development toolkit. https://developer.nvidia.com/cuda-toolkit
- **Git** and **Github** - Version control software, with online repository.
- **lldb** – Open source debugger for detecting errors on MAC OS. https://lldb.llvm.org/
- **MATLAB 2016a** – Program that was used for prototyping of the Sparse Approximation routines. https://www.mathworks.com/products/matlab.html
- **NSight Eclipse Edition** – IDE that was used for development of the CUDA files on Linux. https://developer.nvidia.com/nsight-eclipse-edition
- **Nvprof** – Tool that was used for timing kernel launches in CUDA applications. Distributed with the CUDA development toolkit. https://developer.nvidia.com/cuda-toolkit
- **OpenCV** – Provide libraries for image loading and manipulation in C++. http://opencv.org/
- **Valgrind** – For detecting memory leaks and access violations. http://valgrind.org/
- **Visual Studio 2015 Professional Edition** – IDE and debugger that was primarily used for debugging C++ MEX files on windows. https://www.visualstudio.com/
- **XCode 8** – IDE with debugging environment that was used for debugging C++ MEX files on MAC OS. https://developer.apple.com/xcode/

10.6 Appendix 6 – Start Up Guide

10.6.1 Running MATLAB routines

Four MATLAB scripts have been provided for testing OMP3D, SPMP3D, OMP2D and SPMP2D. Parameters within the scripts can be configured to acquire different results. For example, the pss value can be modified which will cause the routines to break once the specified PSNR value has been reached. The name of the scripts are as follows:

Test_OMP3D_Dan

Test_SPMP3D_Dan

Test_OMP2D

Test_SPMP2D

All of the MATLAB functions have been provided so they should work when called from the working directory.

10.6.2 MEX Application

The C++ files that have a suffix _mex are capable of being compiled using MATLABs mex compiler. In order to do this, the command 'mex' followed by the path of the C++ file and flags for any required libraries is run, as follows:

```
mex Mex/SPMP3D_mex.cpp -lmwblas
```

The routine can then be called by using the name of the file as you would a function in MATLAB. For ease, the test scripts mentioned in Section **10.6.1** have a 'usemex' flag which will tell MATLAB to use the mex file when set as '1'.

10.6.3 CUDA Application

The CUDA application has three dependencies:

- Hardware requirement that the computer must have a NVidia CUDA enabled GPU.
- The computer must have a version of NVidia CUDA Toolkit which can be found in Appendix 5.
- The computer must have the OpenCV Library which can also be found in Appendix 5.

Assuming the dependencies have been installed. The CUDA source files can then be compiled by using the following command. It may be different depending on platform.

```
nvcc SPMP3D.cu h_Matrix.cpp -L/usr/local/lib  -
I/usr/local/include/opencv -I/usr/local/include/opencv2 -
lopencv_features2d -lopencv_imgproc -lopencv_highgui -
lopencv_core -lopencv_imgcodecs -lopencv_videoio  -
lcublas_device -lcudadevrt -lcublas -o SPMP3D.out -rdc=true -
m64 -arch=sm_35 --cudart static -x cu
```

The command can then be executed by:

```
./SPMP3D.out <Image Path>
```