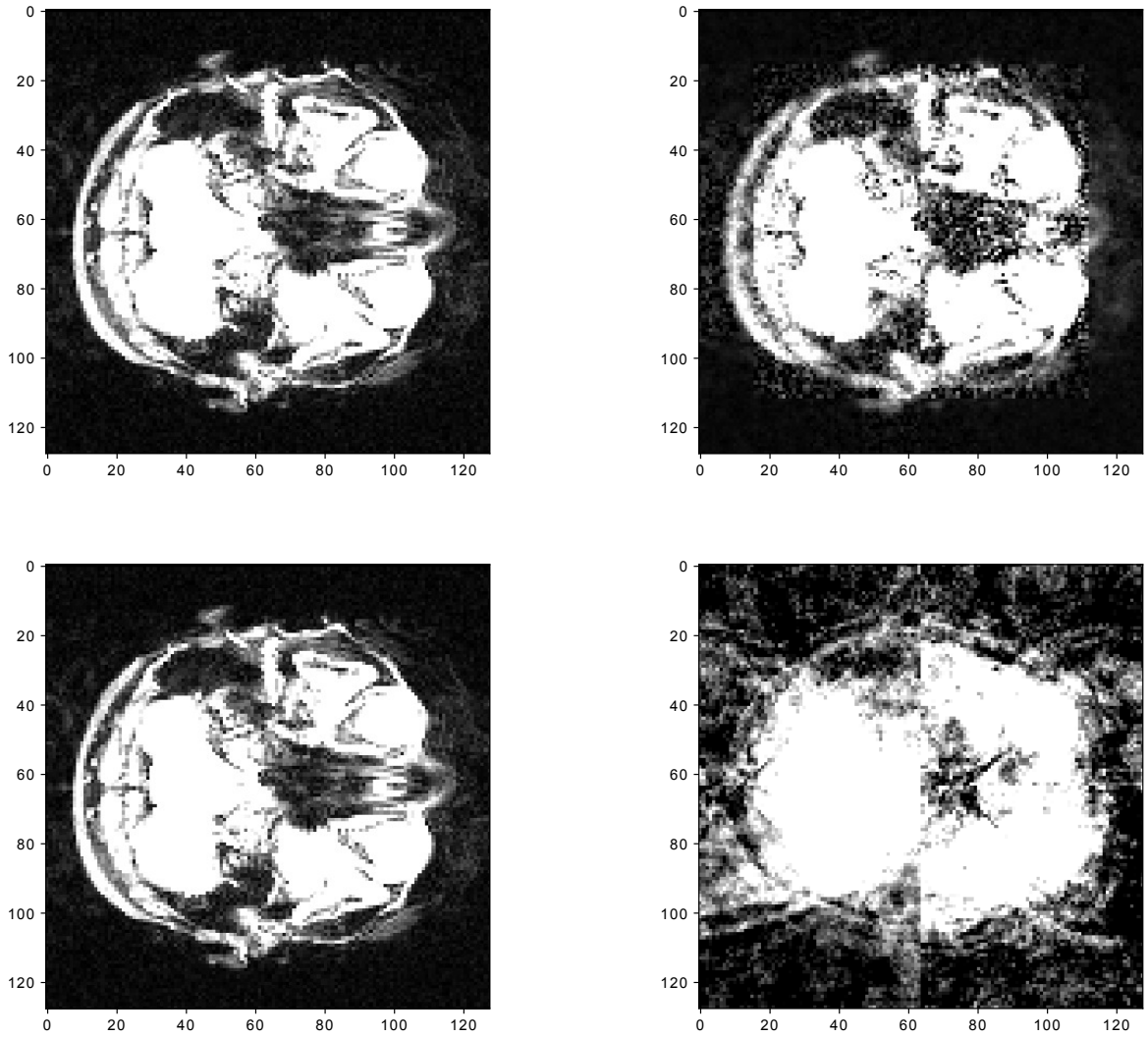
Sequential and Parallel Optimization of a Dictionary Learning Algorithm



P8 Project
Group 18GR871
Electronics and IT
Aalborg University
July 6, 2018

AALBORG UNIVERSITY
STUDENT REPORT

Electronics and IT

Fredrik Bajers Vej 7

9220 Aalborg Ø

http://www.es.aau.dk

Title:

Sequential and Parallel Optimization of a Dictionary Learning Algorithm

P8 project:

Scientific Computing: Dictionary learning with parallel/distributed calculation

Project period:

February 2018 - May 2018

Project group:

18GR871

Participants:

Amalie Vistoft Petersen
Jacob Theilgaard Lassen
Niels Rymann Munthe
Sebastian Biegel Schiøler

Supervisor:

Thomas Arildsen

Number of pages: 157 (total numbers incl. appendices)
Finished: July 6, 2018

Abstract:

The content of this report describes the process of optimizing the implementation of the dictionary learning algorithm Iterative Thresholding and K Residual Means (ITKrM).
First an investigation is made into relevant areas of study regarding machine learning, mainly with a focus on the subject dictionary learning. Hereafter the math behind ITKrM is shown, and shortly described. It is then implemented as close as possible to its mathematical description. Then an iteratively optimization process is made. This is done for sequential optimization on a CPU and parallel optimization for a CPU and a GPU. The implementations are made and tested in Python.
It is concluded that all three implementation methods provided a significant decrease in the execution time of the ITKrM algorithm. The decrease in execution time did come at a prize of an increase in difficulty when implementing.

Preface

This second-semester master project has been developed during the spring of 2018. The project was completed by four students on the Signal Processing and Acoustic master, with a specialization in Signal Processing and Computing at Aalborg University.

In the report, sources are given a number related to the order in which they appear. Each number corresponds to the number shown in the bibliography at the end of the report. Figures with no source are made by the authors of the report. Folders containing the related code for each appendix are included with the report.

The report is separated into 5 parts. The first part is the pre-analysis which goes into introducing dictionary learning, Orthogonal Matching Pursuit (OMP), the ITKrM algorithm, both a mathematical description and an implementation. Furthermore, it looks into which hardware that can be used and defines a problem statement. The second part describes the sequential and parallel optimization, where a sequential optimization, parallel optimization on a CPU and an optimization combining the GPU and CPU is described. The third part goes into the visual performance of the dictionary algorithm and a comparison between the different implementations. The fourth part is the discussion and conclusion. The final part contains the appendices which consist of test and mathematical descriptions.

AAU, May 25, 2018

Amalie Vistoft Petersen
<apet13@student.aau.dk>

Jacob Theilgaard Lassen
<jlasse14@student.aau.dk>

Niels Rymann Munthe
<nmunth14@student.aau.dk>

Sebastian Biegel Schiøler
<sschia14@student.aau.dk>

Contents

1 Introduction **1**
1.1 Initial Problem Statement 2
1.2 Notation 2

Part I Pre-analysis **5**

2 Dictionary Learning **7**
2.1 General Description 8

3 Orthogonal Matching Pursuit **11**

4 ITKrM Dictionary Learning Algorithm **13**
4.1 Algorithm Description 13
4.2 Naïve Implementation 14
4.3 Verification of Naïve ITKrM Implementation 15

5 Hardware **19**
5.1 Processing Units 19
5.2 Compute Cluster 22

6 Pre-analysis Conclusion **25**
6.1 Problem Statement 25

Part II Sequential and Parallel Optimization **27**

7 Sequential Optimization **29**
7.1 First Iteration 29
7.2 Second Iteration 30
7.3 Third Iteration 32
7.4 Discussion 34

8 Parallel Optimization using CPU **35**
8.1 The Multiprocessing Library 35
8.2 Techniques for Examining the Exploitation of Parallelism in the ITKrM Algorithm 36
8.3 First Iteration 37
8.4 Discussion of the Parallel Optimization using CPU 42

9 Parallel Optimization using GPU **45**
9.1 Interacting with a GPU for Numerical Calculations 46
9.2 Initial Test of the GPU 48
9.3 Performance Evaluation 49
9.4 Naïve GPU implementation of ITKrM 50
9.5 Custom CUDA Kernel Implementation 53
9.6 Second Iteration of Custom CUDA Kernel Implementation 59
9.7 Discussion of the Parallel Optimization using GPU 65

Part III Test and Comparison **67**

10 Comparison between Implementation Strategies **69**

11 Visual Performance **73**
11.1 Discussion of the Visual Performance 78

Part IV Discussion and Conclusion **79**

12 Discussion **81**

13 Conclusion **83**

Bibliography **85**

Part V Appendices **89**

A Naïve Implementation: Structural Similarity **91**
A.1 Purpose 91
A.2 Computer Specifications 91
A.3 Preliminary 91
A.4 Procedure 92
A.5 Results 92

B ITKrM vs. DCT: Structural Similarity **95**
B.1 Purpose 95
B.2 Computer Specifications 95
B.3 Preliminary 95
B.4 Procedure 96
B.5 Results 96

C Sequential Optimization - Iteration 0, 1, 2 and 3 **99**
C.1 Purpose 99

C.2 Computer Specifications . . . 99
C.3 Preliminary . . . 99
C.4 Procedure . . . 100
C.5 Results . . . 100

D Sequential Optimization: Memory Usage 105
D.1 Purpose . . . 105
D.2 Preliminary . . . 105
D.3 Procedure . . . 106
D.4 Results . . . 106

E Parallel CPU Optimization - Iteration 1 109
E.1 Purpose . . . 109
E.2 Computer Specifications . . . 109
E.3 Preliminary . . . 109
E.4 Procedure . . . 110
E.5 Results . . . 111

F Speed-up using Multiprocessing 115
F.1 Purpose . . . 115
F.2 Computer Specifications . . . 115
F.3 Preliminary . . . 115
F.4 Procedure . . . 116
F.5 Results . . . 116

G GPU Preliminaries 119
G.1 Purpose . . . 119
G.2 Computer Specifications . . . 119
G.3 Preliminary . . . 119
G.4 Procedure . . . 119
G.5 Results . . . 120

H Parallel GPU Optimization - Naïve 125
H.1 Purpose . . . 125
H.2 Computer Specifications . . . 125
H.3 Preliminary . . . 125
H.4 Procedure . . . 126
H.5 Results . . . 127

I LU Decomposition for Solving a System of Linear Equations 133
I.1 Crout decomposition . . . 133
I.2 Solve Linear System of Equations Implementation . . . 134

J Evaluation of Custom CUDA Kernel for Matrix and Vector Projections **137**
J.1 Purpose . . . 137
J.2 Computer Specifications . . . 137
J.3 Preliminary . . . 137
J.4 Procedure . . . 138
J.5 Results . . . 139

K Evaluation of Custom CUDA Kernels for ITKrM **143**
K.1 Purpose . . . 143
K.2 Computer Specifications . . . 143
K.3 Preliminary . . . 143
K.4 Procedure . . . 144
K.5 Results . . . 145

L Visual Performance **153**
L.1 Purpose . . . 153
L.2 Computer Specifications . . . 153
L.3 Preliminary . . . 153
L.4 Procedure . . . 154
L.5 Results . . . 154

1 | Introduction

In recent years machine learning has seen a surge in popularity due to the increase in availability of larger amounts of data with more variables. This is also referred to as Big Data. Examples of applications in which machine learning are used are natural language processing and recommendation systems. In natural language processing, it is used to recognize speech, this ability is used by companies such as Google and Apple in their products Google Assistant and Siri, making it possible for an electronic device to understand what a user says. In companies such as Netflix, Amazon, and Facebook, machine learning is also used for learning the customers' habits and behaviors. Enabling the possibility of customizing the product to the customer, with the goal of making the product more attractive to the individual person. [1]

There exist two main categories in machine learning, supervised learning and unsupervised learning. Where the distinction between the two subjects is the availability of feedback. In supervised learning, feedback is available as a model is trained by using input data together with a related known output. A trained model can then be used to predict a certain result even in the presence of noise. In unsupervised learning, the model does not get any feedback and tries to find hidden patterns and structures in the input data without knowing a related output. [2]

There are different approaches when finding a suitable model. One approach is called sparse dictionary learning. Here the goal is to find a sparse representation of some input data. This is done by training a dictionary matrix with a sizable amount of data, creating a matrix containing certain features related to the training data. These features are referred to as atoms. The training of the dictionary matrix creates the possibility of approximating the input by a linear combination of atoms. By not using the whole dictionary matrix but only atoms with a certain significance for the specific input it will be possible to find a sparse representation of the input. Dictionary learning has a lot of applications which include areas such as image and audio processing. In these areas, dictionary learning could, for instance, be used for denoising and compression. [3]

This project will look into implementing an existing sparse dictionary learning algorithm with a focus on optimizing the implementation in a parallel or distributed fashion. The existing algorithm used is the algorithm made by Karin Schnass called Iterative Thresholding and K Residual Means (ITKrM) [4]. When training the dictionary matrix it is necessary to use a large amount of data and iterate over the training algorithm numerous times. This results in a high execution time making a parallel and distributed implementation favorable. Through the project a naïve implementation of the ITKrM algorithm will be made and followed by a look into optimizing the algorithm for specific platforms. Here the focus is to implement the algorithm in a parallel fashion. The different implementations will be compared by looking into parameters such as execution time and memory usage.

1.1 Initial Problem Statement

The purpose of this project is to implement a dictionary learning algorithm in Python that exploits inherent parallelism and/or other features for a faster computation or reduced memory usage.

It is necessary to acquire insight into the following, in order to design an implementation that achieves this:

- What is dictionary learning and what is it used for?
- Which algorithm is used and how does it work?
- How can a naïve implementation of the algorithm be implemented and validated?
- Which types of hardware is suitable for implementing the algorithm?

1.2 Notation

A notation section is made to ease the readers understanding of the project. The notations might be explained again through the project, but all notation explanations are also found here.

Variables are written with a cursive font such as K or n. This rule has two exceptions. When describing dimensions of an image signal the variables are not written with a cursive font, e.g. "the execution time of multiplying two N $\times$ N matrices" or "the execution time of multiplying two N by N matrices". The other exception is when the variable comes from a piece of code. An example of this could be: "As nTrainingData is increased", where the variable in this case is nTrainingData. This is the general notation for variables, function, matrices e.t.c. that is a part of a piece of code.

To clarify the difference between variables, vectors, and matrices, vectors are written with a bold cursive font such as $\boldsymbol{y}$, and matrices are with capital letters with a bold cursive font such as $\boldsymbol{D}$. When defining the shape of a vector the notation $\boldsymbol{y} \in \mathbb{R}^M$ describes a column vector with M rows containing all real elements. The shape of a matrix is defined as $\boldsymbol{D} \in \mathbb{R}^{M \times K}$, meaning a matrix with M rows and K columns containing all real elements. The notation regarding the individual columns of a matrix the following notation is used: $\boldsymbol{D} = [\boldsymbol{d}_1, \boldsymbol{d}_2, ..., \boldsymbol{d}_K]$ where $\boldsymbol{d}_1$,$\boldsymbol{d}_2$ up to $\boldsymbol{d}_K$ are columns in $\boldsymbol{D}$. A similar but different notation is when a variable changes between a number of different values. This is written as $n = [1, 10, 100]$, which means that at one point $n = 1$, at a second point $n = 10$, and at a third point $n = 100$.

Through the project, the operators $P()$ and diag() are used. $P()$ is the projection of either a matrix or a vector and is defined as $P(\boldsymbol{A}) = \boldsymbol{A}\left(\boldsymbol{A}^T\boldsymbol{A}\right)^{-1}\boldsymbol{A}^T$. The operator

diag() forms the diagonal matrix for either a matrix or a vector.

In some figures, the horizontal axis is labeled "Dimension [N $\times$ N]". In these figures, a specific value on the horizontal axis is only the value of N, not [N $\times$ N]. The notation [N $\times$ N] is only to clarify that the matrices or images are square.

In the project a notation such as "numpy.* to cupy.*" is used. The star represents an undefined function such as "numpy.zeros to cupy.zeros".

Iterations are a part of the mathematical explanations in various parts of the project. Thus a notation is needed for when a previous vector is used to calculate a new vector, which overwrites the previous vector. This will be written as: $\bar{\boldsymbol{d}} = 3 \cdot \boldsymbol{d}$, where $\bar{\boldsymbol{d}}$ is the new vector.

Part I

Pre-analysis

2 | Dictionary Learning

To further help argue for the importance of dictionary learning, a short introduction to some applications that use dictionary learning will be exemplified.

A field where dictionary learning has proven useful is medical imaging, where signals such as electroencephalogram (EEG) and Magnetic Resonance Imaging (MRI) are of interest. In such areas, it is important to be able to determine the underlying physical causes for the observed signals. This introduces some requirements for the signals such as a limited noise level. If it is not possible to obtain signals with high Signal Noise Ratio (SNR) dictionary learning can be used for denoising the signal. Furthermore, it has been shown that by training a dictionary from MRI scans of breast tissue it is possible to obtain sufficient signals by using the dictionary to reconstruct signals from the less costly ultrasound tomography (UST) scanner. [5]

Super-Resolution (SR) in general is a field where dictionary learning is a valid method. The idea of SR is that from an obtained low-resolution image a high-resolution image can be reconstructed. As mentioned this application is useful in the medical field but is also applicable in space research fields where images from satellites are often of low-resolution. In such a case dictionary learning can be used for enhancing the resolution of such images to improve the circumstances for analysis. [6]

Dictionary learning is applicable within many areas and can be used with various signals such as image and sound signals. For some applications, it is beneficial to use the joint analysis of multiple signals. Such an application could be for speech, where both an audio signal of what is said and a visual signal of the lips moving are available. For such an application dictionary learning has proven to be able to form a proper joint subspace for multiple signals i.e. a multimodal dictionary. Similarly, dictionary learning can be used for multiview imaging, where observations from different viewpoints of a three-dimensional (3D) scene are used to form a dictionary. [5]

Through many years compression of a various range of signals has been made with transformed-based coding such as the Discrete Cosine Transform (DCT) in Joint Photographic Experts Group (JPEG). Dictionary learning has in recent years shown promising results in compressing images. The advantage when using dictionary learning is that the compression can be made especially good for a specific kind of images, e.g. faces. [7]

Face recognition can be made in various ways and with various methods, where dictionary learning are among these methods. A dictionary is trained from a number of training image signals of different faces. When a new image signal of a face is obtained, it is, together with the dictionary, used to obtain a sparse representation of the new face. Then if the new face is recognized, the sparse representation will have non-zero entries concentrated mostly on one of the training images. If the new face is not recognized, the

sparse representation will consist of coefficients from several training images. [8]

2.1 General Description

This section aims to describe the concept of dictionary learning. Let $\boldsymbol{y}_n \in \mathbb{R}^M$ be the full representation of a signal and let $\boldsymbol{x}_n \in \mathbb{R}^K$ be a sparse representation of $\boldsymbol{y}_n$. The idea in dictionary learning is that $\boldsymbol{y}_n$ can be described as a linear combination of $\boldsymbol{x}_n$ and a dictionary $\boldsymbol{D} = [\boldsymbol{d}_1, \boldsymbol{d}_2, ..., \boldsymbol{d}_K] \in \mathbb{R}^{M \times K}$. This description is called the sparse model and is shown in Equation (2.1) [9].

$$\boldsymbol{y}_n = \boldsymbol{D}\boldsymbol{x}_n \tag{2.1}$$

In the sparse model $||\boldsymbol{x}_n||_0 << K$, where $||\cdot||_0$ returns the number of non-zero entries of its argument, i.e. $\boldsymbol{x}_n$ is a sparse representation of $\boldsymbol{y}_n$. The columns of $\boldsymbol{D}$ are the elementary signals also called the atoms. Figure 2.1 shows an illustration of Equation (2.1).

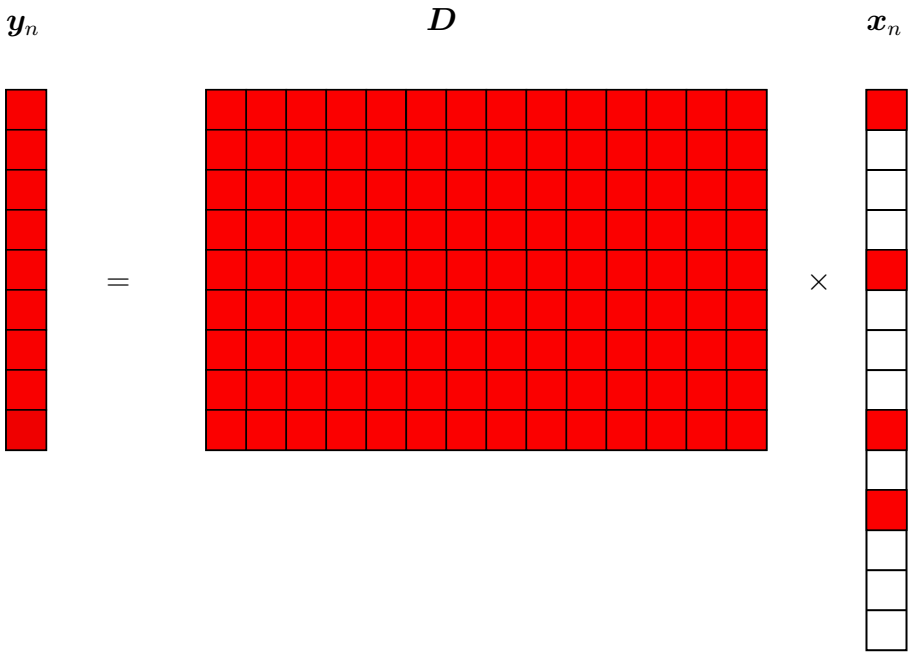


Figure 2.1: Illustration of the sparse model.

In Figure 2.1 the white entries of $\boldsymbol{x}_n$ are zero.

2.1.1 Ways of constructing a dictionary

There are two main categories of topologies for constructing a dictionary: An analytic and a learning based approach. In the analytic approach, the conversion from the full representation to the sparse representation is done by the use of functions instead of the use of training data. Some of the analytic approaches are the Fourier transform, the Wavelet transform and the Hadamard transform [10]. The learning based approaches relates more to the concept of dictionary learning, where an actual dictionary matrix $\boldsymbol{D}$

is trained from a number of training signals. Among the learning based approaches are Principal Component Analysis (PCA), Method of Optimal Directions (MOD) and ITKrM. The learning based approaches are more fitted to the data than the analytic approaches, thus making it perform better in many applications [10]. This increased performance comes at a prize of a heavier computational load, especially for large dimensions of $\boldsymbol{y}_n$ and large n.

3 | Orthogonal Matching Pursuit

Section 2.1 describes the concept of dictionary learning, including the sparse representation $\boldsymbol{x}_n$. To find $\boldsymbol{x}_n$ is in itself a optimization problem [11]. To be able to test the quality of the dictionaries that are produced by the algorithm, it will therefore be necessary to solve the problem of finding $\boldsymbol{x}_n$. The problem of finding $\boldsymbol{x}_n$, with a given error ϵ allowed, can be described as an optimization problem as shown in Equation (3.1).

$$\begin{aligned} &\text{minimize} && \|\boldsymbol{x}_n\|_0 \\ &\text{subject to} && \|\boldsymbol{y}_n - \boldsymbol{D}\boldsymbol{x}_n\|_2 \leq \epsilon \end{aligned} \tag{3.1}$$

The problem is NP-hard [12], and therefore different tricks are needed to be able to find a solution. One way is to relax the ℓ_0-norm to the ℓ_1-norm and then using linear optimization to solve the problem shown in Equation (3.2) [11].

$$\begin{aligned} &\text{minimize} && \|\boldsymbol{x}_n\|_1 \\ &\text{subject to} && \|\boldsymbol{y}_n - \boldsymbol{D}\boldsymbol{x}_n\|_2 \leq \epsilon \end{aligned} \tag{3.2}$$

Other approaches are the greedy strategy approximation methods, which are algorithms that try to approximately solve (3.1). It is chosen for this project to work with the Orthogonal Matching Pursuit (OMP) algorithm, as it is straightforward to implement and does not require solving an optimization problem. The OMP algorithm is shown in Algorithm 1 [13].

Algorithm 1 OMP

1: **Initialize** The residual $\boldsymbol{r}_0 = \boldsymbol{y}_n$ and the subset dictionary $\boldsymbol{S}(c_0) = \varnothing$
2: **For** $i = 0, 1, 2, ...$**do**
3: **Find** $\tau_i = \underset{t}{\operatorname{argmax}} |\boldsymbol{d}_t^T \boldsymbol{r}_i|$
4: **Update** $\boldsymbol{c}_i = \boldsymbol{c}_i \cup \{\tau_i\}$
5: **Let** $\boldsymbol{P}_i = \boldsymbol{S}(\boldsymbol{c}_i) \left(\boldsymbol{S}(\boldsymbol{c}_i)^T \boldsymbol{S}(\boldsymbol{c}_i)\right)^{-1} \boldsymbol{S}(\boldsymbol{c}_i)^T$
6: **Update** $\boldsymbol{r}_i = (\boldsymbol{I} - \boldsymbol{P}_i)\boldsymbol{y}_n$
7: **Until** Stopping condition is achieved
8: **Compute** $\boldsymbol{x}_n = \boldsymbol{S}^+(\boldsymbol{c}_{end})\boldsymbol{y}_n$

The OMP algorithm forms a subset dictionary from the atoms of $\boldsymbol{D}$ which maximizes the inner product between the current atom and the residual. Several conditions can be used as the stopping condition. If a specific sparsity level is wanted, the stopping condition could be an upper bound on the iteration counter i. If a specific quality of the sparse representation $\boldsymbol{x}_n$ is wanted, an error parameter can be defined to decide when to stop the iterations.

4 | ITKrM Dictionary Learning Algorithm

The following chapter studies the dictionary learning algorithm ITKrM. The algorithm is presented and a general description is given. Hereafter a direct implementation of the algorithm is shown. The implementation is verified in different manners to ensure that it is legitimate and that it performs as expected.

4.1 Algorithm Description

The ITKrM algorithm is a sparse dictionary learning algorithm. In [4] it has been proven that the ITKrM algorithm is able to calculate meaningful dictionaries on real data, has a global convergence behavior, and are stable for noisy signals. This algorithm is suitable for parallelization and is robust to noise [4]. For an existing dictionary $\boldsymbol{D} \in \mathbb{R}^{M \times K}$, N training examples $\boldsymbol{y}_n$ and a chosen sparseness level S, an iteration t of the ITKrM algorithm is performed as:

- For all n, find
$$\boldsymbol{I}^t_{\boldsymbol{D},n} = \underset{\boldsymbol{I}:|\boldsymbol{I}|=S}{\arg\max} \|\boldsymbol{D}_{\boldsymbol{I}}^T \boldsymbol{y}_n\|_1 \tag{4.1a}$$
- For all k, calculate
$$\bar{\boldsymbol{d}}_k = \frac{1}{N} \sum_n \left(\boldsymbol{y}_n - P(\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}})\boldsymbol{y}_n + P\left(\boldsymbol{d}_k\right) \boldsymbol{y}_n\right) \cdot \operatorname{sign}\left(\boldsymbol{d}_k^T \boldsymbol{y}_n\right) \cdot \chi\left(\boldsymbol{I}^t_{\boldsymbol{D},n}, k\right) \tag{4.1b}$$
- Output
$$\bar{\boldsymbol{D}} = \left[\frac{\bar{\boldsymbol{d}}_1}{\|\bar{\boldsymbol{d}}_1\|_2}, \frac{\bar{\boldsymbol{d}}_2}{\|\bar{\boldsymbol{d}}_2\|_2}, \cdots, \frac{\bar{\boldsymbol{d}}_K}{\|\bar{\boldsymbol{d}}_K\|_2}\right] \tag{4.1c}$$

where
$$\chi\left(\boldsymbol{I}^t_{\boldsymbol{D},n}, k\right) = \begin{cases} 1 & \text{for } k \in \boldsymbol{I}^t_{\boldsymbol{D},n} \\ 0 & \text{otherwise} \end{cases} \tag{4.1d}$$

Equation (4.1a) finds the index of S atoms in $\boldsymbol{D}$. Atoms are represented by the column vectors of $\boldsymbol{D}$. The atoms that are found through the maximization is the atoms which have the highest correlation to n-th training example. This is done as a high correlation means that those specific atoms, are the S atoms that for the current dictionary can best represent the n-th training example. Equation (4.1b) updates the k-th atom using only the training examples where the k-th atom is in the n-th index set of $\boldsymbol{I}^t_{D,n}$. The dictionary is then normalized in Equation (4.1c).

4.2 Naïve Implementation

The algorithm in Equation (4.1) is implemented with readability in mind and is seen in Listing 4.1.

Listing 4.1: Naïve implementation of ITKrM algorithm.

```
#Algorithm
D_old = D_init
for t in range(maxitr):
    for n in range(N):        # For all n
        I_D[:,n] = max_atoms(D_old, Y[:, n], S)
    D_new = np.zeros((M, K))
    for k in range(K):        # For all k
        for n in range(N):    # Sum all n
            matproj = proj(D_old[:, I_D[:, n]]) @ Y[:, n]    # P(D_I_Dn)
            vecproj = proj(D_old[:, k]) @ Y[:, n]            # P(D_k)
            signer = np.sign(Y[:, n].T @ Y[:, n])            # Sign()
            indicator = np.any(I_D[:, n] == k)            # X()
            D_new[:, k] = D_new[:, k] + (Y[:, n] - matproj + vecproj) * signer * indicator

    scale = np.sum(D_new * D_new, axis=0)
    iszero = np.where(scale < 0.00001)[0]
    D_new[:, iszero] = np.random.randn(M, len(iszero))   # replaces empty atoms

    D_old = normalize_mat_col(D_new)  # Normalizes the dictionary
```

This implementation follows the algorithm in Equation (4.1) almost to a tee with no regard for execution speed or memory usage. The only exception is at the end of the $t-$th iteration. Here there is a possibility that some atoms are completely empty if k is not in any of the index sets $\boldsymbol{I}^t_{\boldsymbol{D},n}$. These empty atoms are then replaced with new random atoms. There is no mention of this step in the algorithm, however, the author of [4] introduces this step in an implementation of the ITKrM algorithm [14, Entry Jan17]. This is to avoid dividing by zero when the dictionary is normalized. This step has therefore been included.

It should be noted that the way the images are fed to the algorithms is by first being cut into smaller pieces with dimensions N_subpic by N_subpic, which defines the dimension of the square subpictures the picture is cut into. This means that in this case the original 32 by 32 pictures are made into subpictures of 16 by 16, and then shaped to 256 entries long vectors. The reason for making these subpictures is that the algorithm tries to find features in the vectorized pictures that are fed to it. So when a full-size picture is fed to the system it will try to derive those features, which will be very specific to that picture. This means if a dictionary is trained with pictures of horses that are centralized, it will have a hard time trying to recreate a picture of a non-centralized horse. When instead these subpictures are made, smaller features a caught, e.g. a head shape. This can then easily be placed wherever it is needed, be it in the center of the picture or at the edge of the picture. It is also possible to cut the pictures into too small subpictures because at a certain point the subpicture features more or less becomes pixel meaning a very fine

reconstruction can be done, but $\boldsymbol{x}_n$ will not be sparse. Also, making subpictures puts in some overhead, as it is needed both to cut the picture into pieces, but also at the other end reassemble it.

4.3 Verification of Naïve ITKrM Implementation

The author of [4] has made a toolbox with an implementation of the ITKrM algorithm [14, Entry Jan17] which the naïve implementation will be compared to. Both algorithms will be fed with the same training data from [15]. All random numbers generated in the algorithm will be taken from the same set of random values to ensure randomness has no influence when comparing the two. This means that the only difference in the resulting dictionaries is caused by the implementation of the algorithm and not external factors.

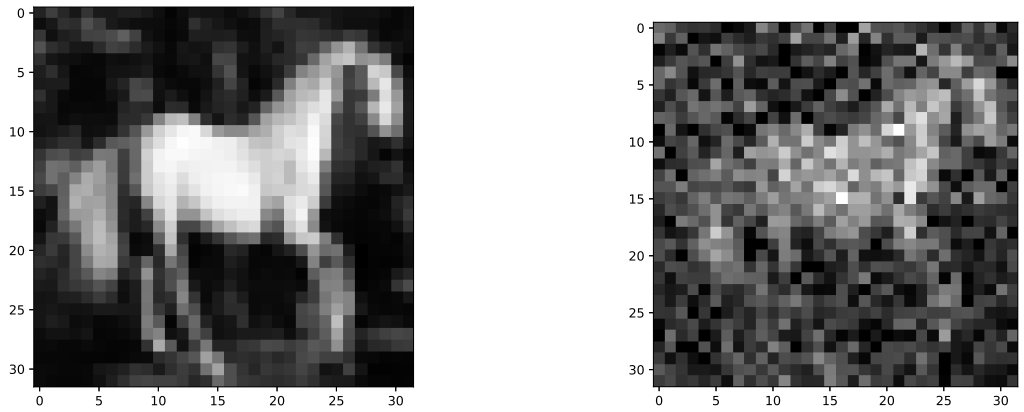
The implementations do not yield identical dictionaries when using the first 10 images of horses in training batch 1 from [15], $S = 16$, $K = 200$, and 20 outer iterations of t. This suggests that the two implementations are different. The ITKrM toolbox is studied in detail to find out why the implementations do not give identical results. It is found that the vector projection $P(\boldsymbol{d}_k) = \boldsymbol{d}_k(\boldsymbol{d}_k^T\boldsymbol{d}_k)^{-1}\boldsymbol{d}_k^T$ is calculated as $P(\boldsymbol{d}_k) = \boldsymbol{d}_k\boldsymbol{d}_k^T$. If the projection is implemented similarly in the naïve implementation then the two implementations produce identical dictionaries with the previous setup. No changes to the variables, test batch or subjects resulted in variations between dictionaries. This means the implementation performs the algorithm correctly. The author of [4] argues that the new vector projection is a fair transformation in [16, p. 5-6] when assuming the signals are perfectly sparse in a generating dictionary $\boldsymbol{D}$. This is not the case given the natural images in the training data [15] and will not be a part of the naïve implementation. The algorithm is still verified as they produce identical results when doing the same math.

There is however not always a possibility to check an implementation by comparison, so another way to verify the naïve implementation is to retrieve sparse $\boldsymbol{x}_n$ from the dictionary and checking whether the trained dictionary produces a better sparse solution than a random dictionary or an analytical dictionary.

4.3.1 Comparison between a Random and a Trained Dictionary

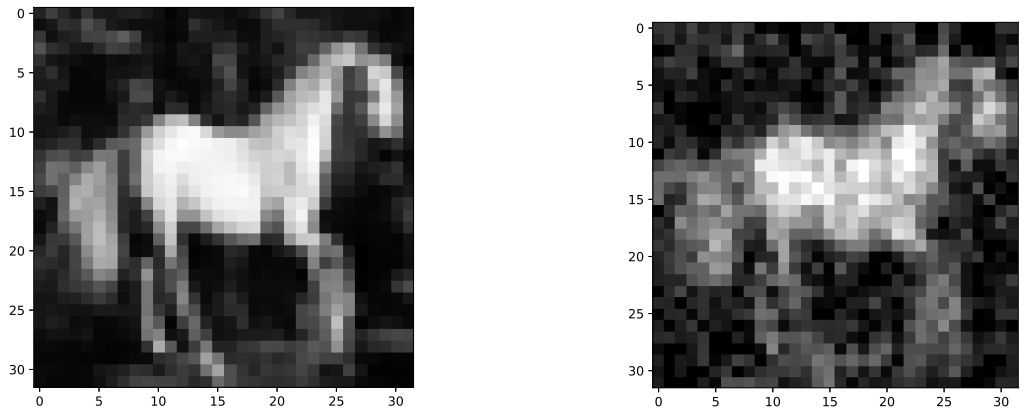
In order to verify the naïve implementation a dictionary trained by the ITKrM algorithm $\boldsymbol{D}_{\text{trained}}$ is compared to a randomly generated dictionary $\boldsymbol{D}_{\text{random}}$. The comparison is made by using an image signal $\boldsymbol{y}$ as input to the OMP algorithm together with either $\boldsymbol{D}_{\text{random}}$ or $\boldsymbol{D}_{\text{trained}}$. From this two sparse $\boldsymbol{x}$ signals are generated, one for the random dictionary $\boldsymbol{x}_{\text{random}}$ and one for the trained dictionary $\boldsymbol{x}_{\text{trained}}$. The variables $\boldsymbol{x}_{\text{random}}$ and $\boldsymbol{D}_{\text{random}}$ are then used to generate the image signal $\boldsymbol{y}_{\text{random}}$, and $\boldsymbol{x}_{\text{trained}}$ and $\boldsymbol{D}_{\text{trained}}$ are used to generate the image signal $\boldsymbol{y}_{trained}$. These are compared to the signal $\boldsymbol{y}$ with respect to the Structural Similarity (SSIM) index, which is a measure of the perceived quality of a digital image. If two image signals are identical the SSIM index equals 1, and for less

identical images the SSIM index decreases [17]. The comparison between the random and the trained dictionary is made in Appendix A, and the results are shown in Figure 4.1 and Figure 4.2, where $\boldsymbol{y}$ is the beforeImage and $\boldsymbol{y}_{\text{random}}$ and $\boldsymbol{y}_{\text{trained}}$ are the afterImage's.



(a) beforeImage. **(b)** afterImage.

Figure 4.1: Results from naïve implementation with random dictionary, with a sparsity level $S = 40$. SSIM index = 0.7787.



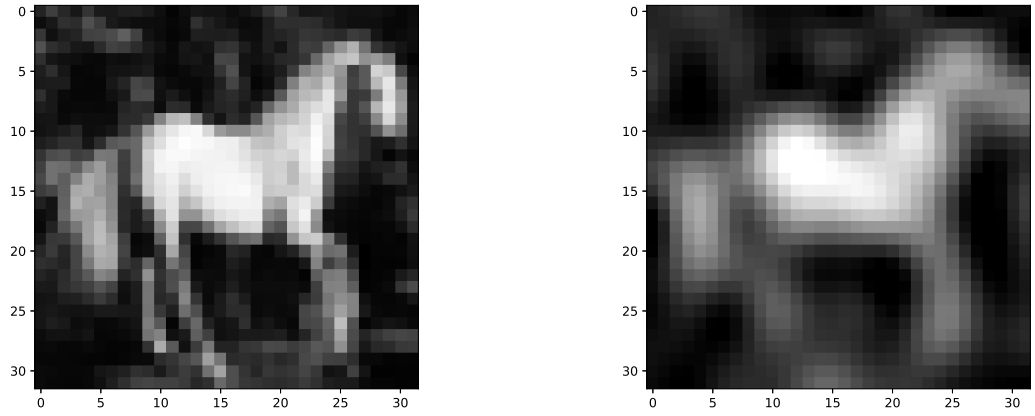
(a) beforeImage. **(b)** afterImage.

Figure 4.2: Results from naïve implementation with trained dictionary, with a sparsity level $S = 40$. SSIM index = 0.9596.

From Figure 4.1 and Figure 4.2 it is seen that the trained dictionary is better at generating sparse representations than a random dictionary, thus verifying the naïve implementation. To futher verify the the algorithm, a comparison with the DCT algorithm is done.

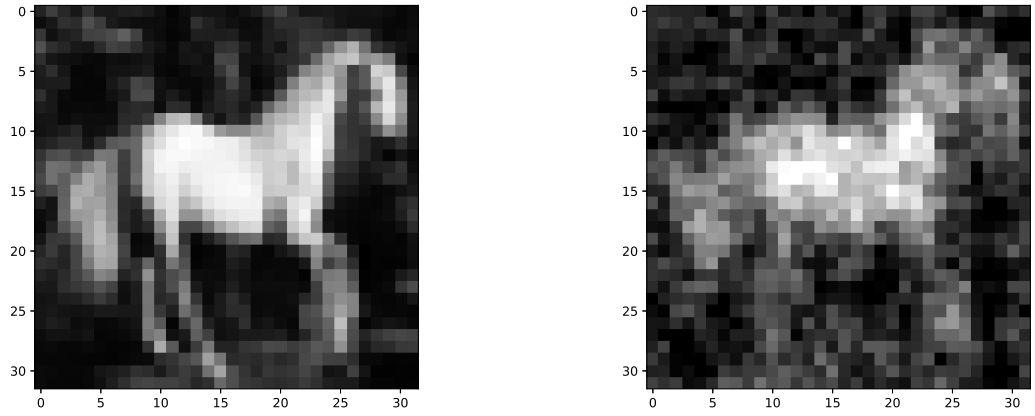
4.3.2 Comparison between DCT and a Trained Dictionary

In order to further verify the naïve implementation a dictionary trained by the ITKrM algorithm $\boldsymbol{D}_{\text{trained}}$ is compared to a picture compressed with DCT. The comparison is made by using an image signal $\boldsymbol{y}$ as input to a DCT algorithm from scipy and as an input to the OMP algorithm together with $\boldsymbol{D}_{\text{trained}}$. The two algorithms are allowed an equal amount of non-zero entries to recreate the image. The two resulting images are then compared to the original signal $\boldsymbol{y}$ with respect to the SSIM index. The comparison between DCT and the trained dictionary is made in Appendix B, and the results are shown in Figure 4.3 and Figure 4.4. Signal $\boldsymbol{y}$ is the beforeImage, and $\boldsymbol{y}_{\text{trained}}$ and the scipy DCT are the afterImage's.



(a) beforeImage. **(b)** afterImage.

Figure 4.3: Results from DCT algorithm. SSIM index = 0.7363.



(a) beforeImage. **(b)** afterImage.

Figure 4.4: Results from naïve implementation with trained dictionary, with a sparsity level S = 25. SSIM index = 0.8984.

From Figure 4.3 and Figure 4.4 it is seen that the trained dictionary is better at generating sparse representations than the DCT algorithm. It should be noted that the execution time of the DCT is faster, and it does not necessarily require a dictionary to be stored. Instead, it is possible to have an algorithm to rebuild the picture from the sent coefficients. In case only a slower computing unit is available it might not be preferred to have a semi-complex algorithm to rebuild the picture, instead making a matrix-vector product would be preferred, as it is done with the trained dictionary.

5 | Hardware

In the following chapter, a small overview is given of some conventional hardware that is suitable for scientific tasks. This includes Central Processing Units (CPUs), Field-Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs). In addition to this, a description is given of the structure of an available compute cluster.

5.1 Processing Units

In the area of scientific computing, a wide selection of platforms are suitable when executing scientific code. Some platforms perform well for specific tasks but are challenged when handling other tasks, and some platforms have a generally decent performance across a wide selection of tasks. For this reason, different platforms are often combined for solving a given problem. Among the platforms for scientific computing are CPUs, FPGAs and GPUs [18]. These three platforms will be investigated with the purpose of being able to make a deliberate choice of how to implement the ITKrM algorithm.

5.1.1 CPU

Among the platforms that perform decently for a wide set of tasks is the CPU. The CPU is a necessary part of modern day computers, thus making it an important part of scientific computing. Among the features of a modern CPU are [19]: Complex instruction flows, wide floating point arithmetic vector units, large amount of shared memory, and fast CPU-CPU interconnects. In Figure 5.1 a simplified high-level block diagram of a Intel Core i7 CPU architecture is shown [20].

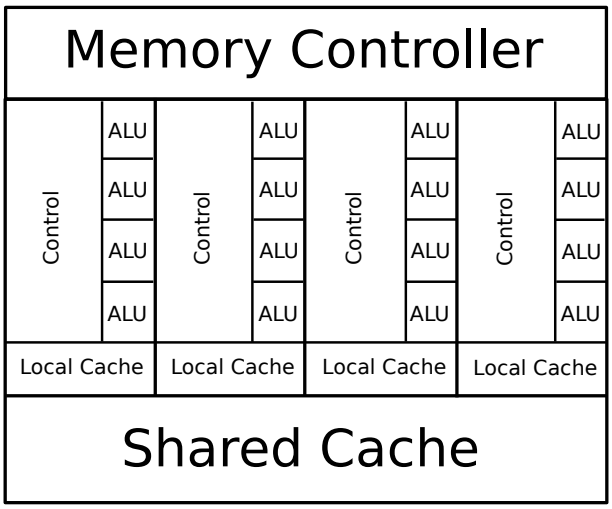


Figure 5.1: Simplified high-level block diagram of a Intel core i7 CPU [20].

The complex instruction flows require a relatively large amount of control logic, which makes the CPU able to perform loops, conditionals, etc. This requirement for control logic leaves less room for memory and computing units such as Arithmetic logic units (ALUs). This limited amount of computing units implies a limited performance in raw floating point operations. A strong feature of the CPU is its fast CPU-CPU interconnects which makes it suitable for networks containing several CPUs.

The versatility is the key strength of the CPU and the possibility of connecting it with coprocessors such as FPGAs and GPUs makes it possible to create a combined system with better performance than a standalone CPU [19].

5.1.2 FPGA

The FPGA is a platform suited for implementing both application specific and domain-specific digital circuits. The FPGA consists of an array of configurable logic blocks (CLBs) which are interconnected through connect boxes that are connected by switch boxes, also called an island architecture [21]. The FPGA architecture is seen in Figure 5.2.

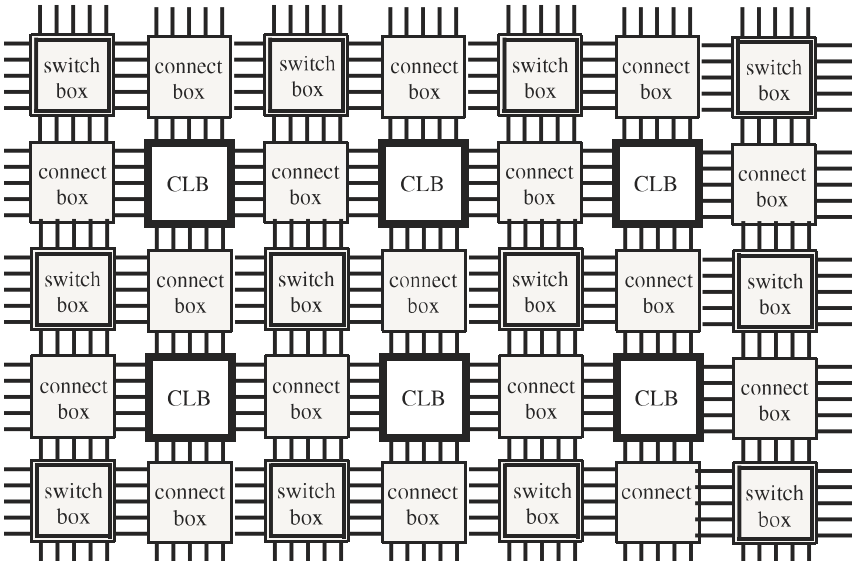


Figure 5.2: Architecture of FPGA [21].

The connect boxes and the switch boxes are the routing resources and are both reconfigurable. The architecture of the FPGA also makes it possible to run more tasks in parallel. These are some of the features that have lead to the use of FPGAs in scientific computing. A few drawbacks of the FPGA are that the clock frequency of a FPGA is substantially lower than in microprocessors, and they are power-hungry and expensive [21].

5.1.3 GPU

Driven by the demands of video games the GPU has been developed to a level where it is suited for a much wider application area than only video games. Among these areas are scientific computing, which uses the strong parallel features of the GPU. The strong parallelism of the GPU is a result of its architecture where the graphics pipeline makes it possible to execute operations separately. Thus the GPU is more suited for heavy arithmetic operations than the CPU. Another aspect of the GPU is that throughput is prioritized higher than latency. The reason for this is that as the GPU was originally developed for graphics, so the latency of the GPU is fast compared to the human visual system [22]. Figure 5.3 illustrates a simplified high-level Fermi NVIDIA GPU architecture [23].

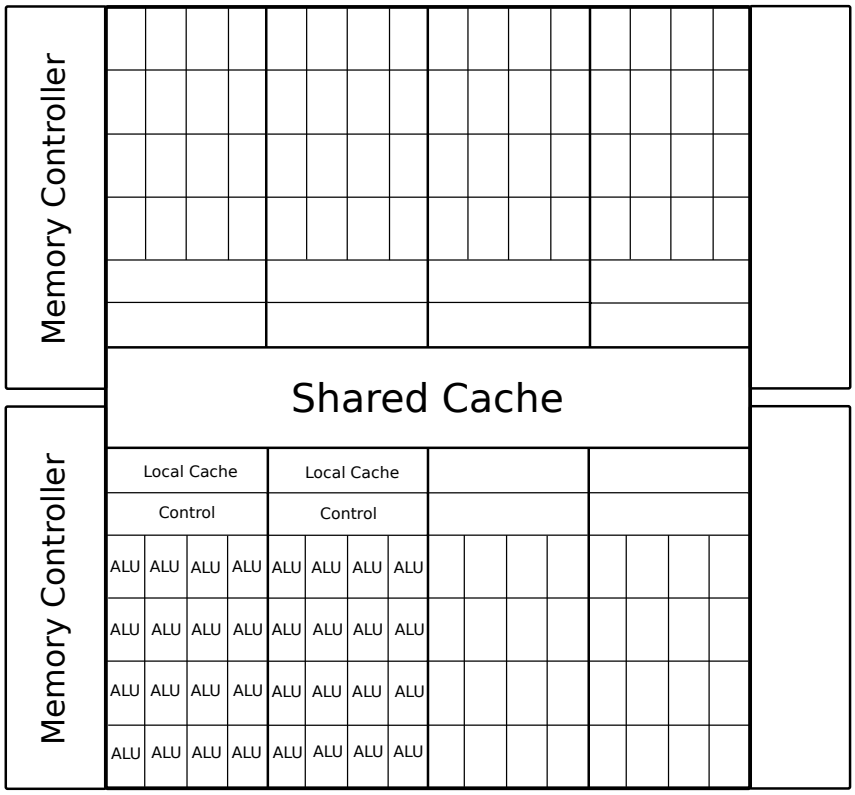


Figure 5.3: Simplified high-level block diagram of a Fermi NVIDIA architecture [23].

A GPU can be categorized as a coprocessor because the amount of control logic is very limited compared to for example a CPU. Thus the GPU is often used in systems alongside different types of computational units e.g. a CPU [19]. Figure 5.4 shows an architecture containing one CPU and two GPUs.

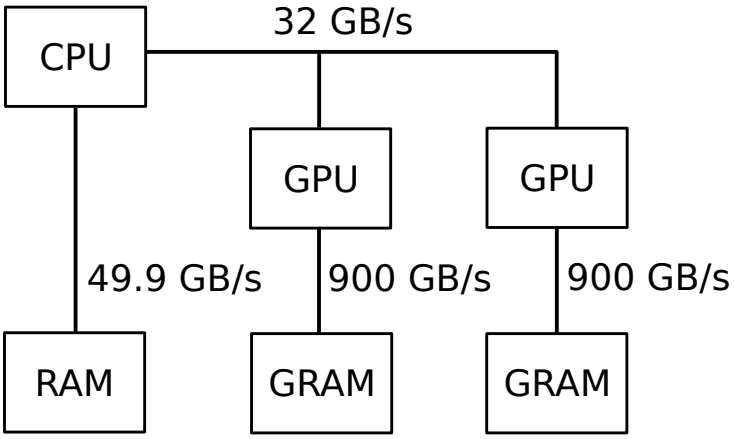


Figure 5.4: Transfer speed between a CPU, GPU, RAM and GRAM [24].

In the figure a Peripheral Component Interconnect Express (PCIe) x16 bus connects the CPU to the GPUs [24]. In Figure 5.4 the transfer rate from the CPU to the GPUs is 32 GB/s. When using such a combined architecture this particular bus should be used carefully to avoid making it a bottleneck.

The three processing units CPU, FPGA and GPU have now been described, with the purpose of being able to make a deliberate choice of how to implement the ITKrM algorithm. A specific processing unit will later be chosen. In the following section, an overview is given of an available compute cluster which will be used in the project.

5.2 Compute Cluster

Students at Aalborg University have the possibility of gaining access to a compute cluster called Birdnest. Birdnest consist of three main segments, these are the Administration nodes, CPU based compute nodes and GPU based compute nodes. A simplified high-level block diagram of the compute cluster can be seen in Figure 5.5.

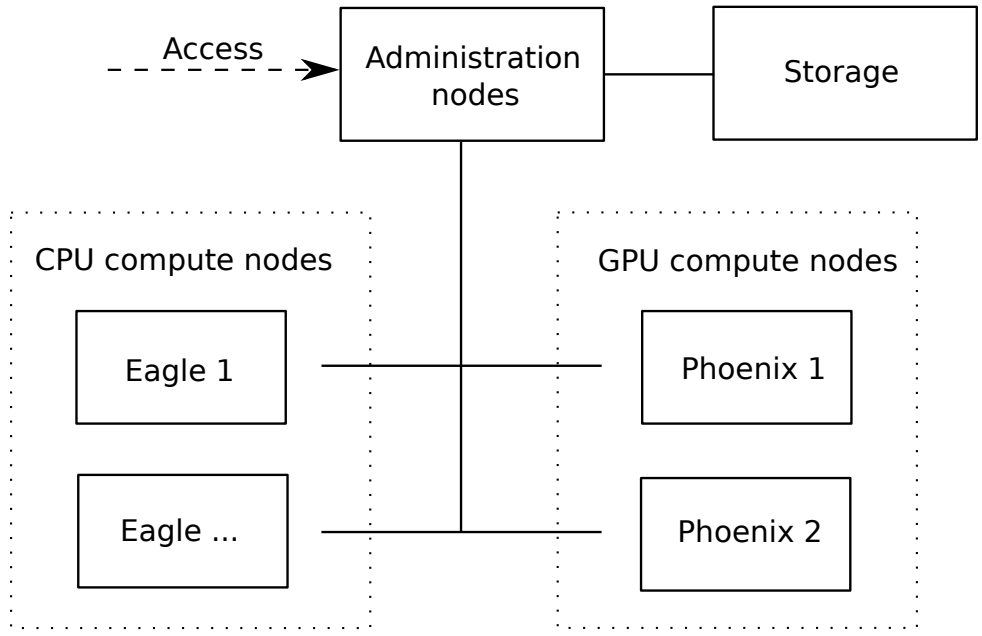


Figure 5.5: Simplified architecture of the compute cluster, with the relevant nodes.

The administration nodes consist of three nodes. One node is a login node available through SSH, which is used for compiling and installing software, movement of data to and from the cluster, and submission of jobs. The second node is the primary file system storage and it hosts the job scheduler and resource manager daemon. The job scheduler and resource manager handles the allocating of compute nodes, allocation of computing resources and the scheduling of jobs. The last node handles the secondary file system backup storage.

The second segment is the CPU compute nodes. This segment consists of multiple Intel CPUs. The CPU compute nodes are five nodes called Eagle. These are five identical configured compute nodes. One Eagle node consist of two 12 core CPUs operating at 2.7 GHz, 384 GiB RAM and 200 GB + 800 GB SSD drive space.

The third and last segment is the GPU compute nodes. The relevant compute nodes in this segment are the Phoenix 1 and Phoenix 2. The Phoenix nodes are based on the Intel Nehalem architecture. Phoenix 1 contains: two 4 core CPUs operating at 2.93 GHz, 96 GiB RAM, 300 GB mechanical hard drive space and 3 NVIDIA GTX580 with 3 GiB memory each. Phoenix 2 contains the same as Phoenix 1 except for the GPUs which are 3 NVIDIA Tesla C2070 with 6 GB memory each.

The three processing unit types, CPU, FPGA and GPU, and a compute cluster has been described. The descriptions will help to make a deliberate choice in the implementation phase of the ITKrM algorithm.

6 | Pre-analysis Conclusion

The areas from the initial problem statement section 1.1 has been investigated through the pre-analysis. The concept of dictionary learning has been clarified and some of the applications of dictionary learning has been presented. A description of the OMP algorithm has been made with the purpose of being able to investigate the quality of a learned dictionary. The ITKrM algorithm was chosen to be the dictionary learning algorithm to be implemented in the project. The algorithm has been described from a mathematical point of view and a naïve implementation has been made of the algorithm. The naïve implementation was made with a focus on readability to clearly illustrate the relation between the mathematical description and the implementation. The naïve implementation was verified using three approaches. One approach was comparing the naïve implementation to a reference implementation, which was made by the inventor of the ITKrM algorithm. The last two approaches were to compare the trained dictionary from the naïve implementation with a randomly generated dictionary and a DCT algorithm. In all cases, the naïve implementation behaved as expected and thus it was concluded that the naïve implementation works properly. A short investigation was made into the processing units typically used in the field of scientific computing, wherein dictionary learning lies. The three processing units CPU, FPGA and GPU was investigated and their advantages and their drawbacks were presented. Furthermore, an introduction to the Birdnest cluster was given.

Based on the investigated subjects in the pre-analysis a problem statement can be made together with some choices regarding the future work of the project.

6.1 Problem Statement

How do different implementation methods and hardware platforms alter the execution time of the ITKrM algorithm and its difficulty of implementation?

To answer the problem statement some decisions have been made. The decisions are as follows:

- The processing units used for the project will only include CPUs and GPUs. This is due to the time limitation on the project.
- Testing will be done on the Phoenix 1 node as far as possible. This is because the Phoenix nodes seem capable to test all implementations and it will make a basis for comparing all the different implementations.
- The implementation of the OMP algorithm will not be a subject for optimization. This is outside the scope of the project, which focuses on the ITKrM algorithm and

it is evaluated that the processes of optimizing the OMP and ITKrM algorithm is very similar.

- The ITKrM algorithm will together with the OMP algorithm be used for compression of image signals. This is seen as a simple way to verify the functionality of the algorithm.

Now that the pre-analysis has been concluded and a problem statement has been made, the optimization of the naïve implementation of the ITKrM algorithm can begin. Here the ITKrM algorithm is optimized sequentially, which is followed by a parallel optimization. The parallel optimization includes both a parallel optimization only on CPUs and a implementation on a combination of CPUs and GPUs.

Part II

Sequential and Parallel Optimization

7 | Sequential Optimization

This chapter details the sequential optimization process of the implemented ITKrM algorithm. The implementation will be optimized in an iterative manner by attempting to optimize the most time-consuming process and hereafter reevaluating the implementation. The reevaluation will contain the confirmation that the new generated dictionary matrix is correct, together with the resulting change in the total execution time. The new dictionary matrix is compared with the dictionary matrix generated by the naive implementation to ensure the new implementation is correct.

After the optimization of the implemented algorithm is finalized, an analysis of the total amount of memory usage is done. The balance between execution time, memory usage and time used on optimizing the code is discussed to conclude the sequential optimization.

All sequential implementations are evaluated w.r.t. execution time with the line_profiler tool to give an overview of the computationally heaviest lines in each implementation. The naïve implementation and the last sequentially optimized implementation are evaluated w.r.t. memory usage with the memory_profiler tool.

7.1 First Iteration

The execution time of each line of code in the naïve implementation is measured in Appendix C. The innermost loop is the most time consuming and is shown in Listing 7.1.

Listing 7.1: Time consumption of innermost loop of the naïve implementation.

```
Timer unit: 1e-06 s
Total time: 2623.16 s
Line no.    Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    50      4020         4264      1.1      0.0  for k in range(K):
    51   1604000      2701154      1.7      0.1      for n in range(N):
    52   1600000   2136411164   1335.3     81.4          matproj = proj(D_old[:,I_D[:,n]])
                                                              @Y[:,n]
    53   1600000    384668443    240.4     14.7          vecproj = proj(D_old[:,k])@Y[:,n]
    54   1600000     19813538     12.4      0.8          signer = np.sign(D_old[:,k].T@Y[:,n])
    55   1600000     38161970     23.9      1.5          indicator = np.any(I_D[:,n]==k)
    56   1600000     40940790     25.6      1.6          D_new[:,k] = D_new[:,k] + (Y[:,n]
                                                                  - matproj + vecproj)
                                                                  *signer*indicator
```

It shows in line 52 and 53 that the matrix projection takes 81.4 % of the total time, the vector projection 14.7 %, and both are executed 1,600,000 times. The main focus of the first iteration is then to reduce the execution time of the matrix projection.

7.1.1 Implementation Optimization

The first thing to notice in the naïve implementation in Listing 4.1, is that the matrix projection does not depend on k, yet it is performed K times as a result of being inside the k-loop. As there are no operations performed between the k- and n-loops their order can be swapped with no impact on the result. The matrix projection can then be moved out of the k-loop.

The second thing to notice is that all operations are executed even if the indicator function equals zero. The number of operations performed can be limited by only performing the operations when the indicator function equals one. This can be done by removing the indicator variable entirely and have the k-loop only execute for $k \in \boldsymbol{I}^t_{\boldsymbol{D},n}$.

These changes are implemented in Listing 7.2.

Listing 7.2: First iteration of the sequential optimization.

```
1 for n in range(N):                                  # Swap k and n loops.
2     matproj = proj(D_old[:,I_D[:,n]])@Y[:,n]        # Moved out of k loop.
3     for k in I_D[:,n]:                              # Only when indicator function = 1
4         vecproj = proj(D_old[:,k])@Y[:,n]
5         signer = np.sign(D_old[:,k].T@Y[:,n])
6         D_new[:,k] = D_new[:,k]+(Y[:,n]-matproj+vecproj)*signer
```

7.1.2 Reevaluation

After the first iteration of optimization, the code is verified to run correctly as the resulting dictionary of the implementation is identical to the dictionary generated by the naïve implementation.

The execution time of the code in Listing 7.2 is found in Listing C.2 where it is seen that the total execution time is reduced from 2623.16 s to 101.30 s. The matrix projection is also no longer the most time-consuming task.

7.2 Second Iteration

The results for the first iteration can be found in Appendix C. The inner loop of n is shown in Listing 7.3. Notice that the line with the variable matproj only takes up 10.9 % of the total time, compared to 81.4 % in the naïve implementation. The largest time consumer at the moment is the line with the variable vecproj which takes up 77.0 % of the total time.

Listing 7.3: Time consumption of innermost loop of the sequential optimization iteration 1.

```
Timer unit: 1e-06 s
Total time: 101.303 s
Line no.    Hits         Time  Per Hit   % Time  Line Contents
============================================================
```

```
50      8020        11058      1.4      0.0  for n in range(N):
51      8000     11026233   1378.3     10.9      matproj = proj(D_old[:,I_D[:,n]])@Y[:,n]
52    328000       750731      2.3      0.7      for k in I_D[:,n]:
53    320000     77962189    243.6     77.0          vecproj = proj(D_old[:,k])@Y[:,n]
54    320000      4115208     12.9      4.1          signer = np.sign(D_old[:,k].T
                                                            @Y[:,n])
55    320000      6962337     21.8      6.9          D_new[:,k] = D_new[:,k]+
                                                              (Y[:,n]-matproj+vecproj)
                                                              *signer
```

The k-loop is removed to decrease the time consumed on line 53. This can be accomplished by exploiting the independent nature of the loop, as no variable is dependent on the former iteration. This could decrease the execution time by removing some overhead of the loop. The operations performed by the line with vecproj is,

$$\text{vecproj} = \boldsymbol{d}_k(\boldsymbol{d}_k^T\boldsymbol{d}_k)^{-1}\boldsymbol{d}_k^T\boldsymbol{y}_n \tag{7.1}$$

where $\boldsymbol{d}_k \in \mathbb{R}^M$ and M is the length of the training example $\boldsymbol{y}_n$. As each iteration in the k loop is independent, it is possible to utilize the matrix $\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \in \mathbb{R}^{M\times S}$ instead of $\boldsymbol{d}_k$. It is important to notice that $(\boldsymbol{d}_k^T\boldsymbol{d}_k)^{-1}$ and $\boldsymbol{d}_k^T\boldsymbol{y}_n$ results in scalars. To calculate the vector projection using matrix operations, these scalars should be multiplied with each column of $\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}$. This is done by forming the diagonal matrices,

$$\begin{bmatrix} (\boldsymbol{d}_{i_0}^T\boldsymbol{d}_{i_0})^{-1} & & \\ & \ddots & \\ & & (\boldsymbol{d}_{i_S}^T\boldsymbol{d}_{i_S})^{-1} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \boldsymbol{d}_{i_0}^T\boldsymbol{y}_n & & \\ & \ddots & \\ & & \boldsymbol{d}_{i_S}^T\boldsymbol{y}_n \end{bmatrix}$$

where i_s is the s-th element in the index set $\boldsymbol{I}^t_{\boldsymbol{D},n}$ and $\boldsymbol{d}_{i_s}$ is the i_s-th column of $\boldsymbol{D}$. These are the inverse of the diagonal of the matrix $\left(\boldsymbol{D}^T_{\boldsymbol{I}^t_{\boldsymbol{D},n}}\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}\right)$ and the vector $\left(\boldsymbol{D}^T_{\boldsymbol{I}^t_{\boldsymbol{D},n}}\boldsymbol{y}_n\right)$ formed as diagonal matrices. Forming a diagonal matrix from the diagonal of a matrix $\boldsymbol{X} \in \mathbb{R}^{L\times L}$ is defined as

$$\text{diag}\left(\boldsymbol{X}\right) = \sum_{l=0}^{L-1} \boldsymbol{e}_l\boldsymbol{e}_l^T\boldsymbol{X}\boldsymbol{e}_l\boldsymbol{e}_l^T, \tag{7.2}$$

where $\boldsymbol{e}_l$ is the l-th unit vector of the standard basis.

Forming a diagonal matrix from a vector $\boldsymbol{x} \in \mathbb{R}^L$ is defined as

$$\text{diag}\left(\boldsymbol{x}\right) = \sum_{l=0}^{L-1} \boldsymbol{e}_l\boldsymbol{e}_l^T\boldsymbol{x}\boldsymbol{e}_l^T, \tag{7.3}$$

and the vector projection is then calculated using matrices in Equation (7.4).

$$\text{vecproj} = \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}\text{diag}\left(\boldsymbol{D}^T_{\boldsymbol{I}^t_{\boldsymbol{D},n}}\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}\right)^{-1}\text{diag}\left(\boldsymbol{D}^T_{\boldsymbol{I}^t_{\boldsymbol{D},n}}\boldsymbol{y}_n\right) \tag{7.4}$$

To make signer independent of k, the matrix $\boldsymbol{D}_{\boldsymbol{I}^t_{D,n}}$ is used instead of the vector $\boldsymbol{d}_k$. The matrix $\boldsymbol{D}_{\boldsymbol{I}^t_{D,n}}$ is also used in D_new and a new matrix is formed with dimensions $M \times S$ containing the n-th training example repeated horizontally S times. The transformed code can be seen in Listing 7.4.

Listing 7.4: Iterative 2 of the sequential optimization.

```
1 for n in range(N):
2     matproj = np.repeat(np.array([ proj(D_old[:,I_D[:,n]])@Y[:,n] ]).T, S, axis=1)
3     vecproj = D_old[:,I_D[:,n]] @ np.diag(np.diag(D_old[:,I_D[:,n]].T @ D_old[:,I_D[:,n]])**-1
              *(D_old[:,I_D[:,n]].T@Y[:,n]))
4     signer = np.sign(D_old[:,I_D[:,n]].T@Y[:,n])
5     D_new[:,I_D[:,n]] = D_new[:,I_D[:,n]] +(np.repeat(np.array([Y[:,n]]).T, S, axis=1)
                        - matproj + vecproj)*signer
```

7.2.1 Reevaluation

The code is verified as the resulting dictionary is identical to the generated dictionary created by the naïve implementation.

The execution time of the code in Listing 7.4 can found in Listing C.3. The total execution time is reduced from 101.30 s to 16.02 s. The matrix projection is once again the most time-consuming task.

7.3 Third Iteration

The results for the second iteration can be found in Appendix C. The inner loop of n is shown in Listing 7.5. The line matproj has increased its percentual execution time from 10.9 % to 72.7 %, but vecproj has decreased from 77.0 % to 11.0 %, making matproj the heaviest line of the program.

Listing 7.5: Time consumption of innermost loop of the sequential optimization iteration 2.

```
Timer unit: 1e-06 s
Total time: 16.0257 s
Line no.  Hits         Time    Per Hit   % Time  Line Contents
==============================================================
50        8020        18069      2.3      0.1    for n in range(N):
51        8000     11645781   1455.7     72.7        matproj = np.repeat(np.array([
                                                            proj(D_old[:,I_D[:,n]])
                                                            @Y[:,n] ]).T, S, axis=1)
52        8000      1770207    221.3     11.0        vecproj = D_old[:,I_D[:,n]] @
                                                            np.diag(np.diag(
                                                            D_old[:,I_D[:,n]].T @
                                                            D_old[:,I_D[:,n]])**-1
                                                            *(D_old[:,I_D[:,n]].T@Y[:,n]))
53        8000       322517     40.3      2.0        signer = np.sign(D_old[:,I_D[:,n]].T
                                                           @Y[:,n])
54        8000      1782121    222.8     11.1        D_new[:,I_D[:,n]] = D_new[:,I_D[:,n]] +
                                                                (np.repeat(np.array([
                                                                Y[:,n]]).T, S, axis=1)
```

```
                                                    matproj + vecproj)
                                                    *signer
```

It is possible to reduce the execution time by looking further into the calculated vecproj and its calculation of $\text{diag}(\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}^T \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}})^{-1}$. Notice that a whole matrix-matrix product is calculated despite only the diagonal of the calculation is needed.

It is also entirely possible that certain columns of $\boldsymbol{D}$ is in the index set $\boldsymbol{I}^t_{\boldsymbol{D},n}$ for multiple training examples. This means that an arbitrary column of $\boldsymbol{D}$ is used for calculating $\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}^T \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}$ more than once for the vector projection. By inspecting the full expression for calculating a new dictionary $\boldsymbol{D}$ in Equation (7.5), it is seen that this is not only a problem for the vector projection.

$$\begin{aligned}\bar{\boldsymbol{D}} = \frac{1}{N}\sum_n \Bigg(\boldsymbol{Y}_n + \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \left(\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}^T \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \right)^{-1} \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}^T \boldsymbol{y}_n \\ - \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \text{diag}\left(\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}^T \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \right)^{-1} \text{diag}\left(\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}^T \boldsymbol{y}_n \right) \Bigg) \text{sign}\left(\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}^T \boldsymbol{y}_n \right)\end{aligned} \tag{7.5}$$

It is seen that the matrix $\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}^T \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}$ is computed twice and could contain columns of $\boldsymbol{D}$ that are used more than once. The probability of an arbitrary column appearing more than once is 100 % when there are more training examples than atoms. This is always the case because N should always be larger than K. This means there is room for reduction in the number of computations to be performed.

This can be done by computing $\boldsymbol{D}^T\boldsymbol{D}$ outside of the n-loop. A submatrix is then extracted inside the loop instead of computing it for each n. This means that the new updated dictionary $\boldsymbol{D}$ is computed by Equation (7.6).

$$\begin{aligned}\bar{\boldsymbol{D}} = \frac{1}{N}\sum_n \Bigg(\boldsymbol{Y}_n + \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \left(\left(\boldsymbol{D}^T\boldsymbol{D} \right)_{\boldsymbol{I}^t_{\boldsymbol{D},n}, \boldsymbol{I}^t_{\boldsymbol{D},n}} \right)^{-1} \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}^T \boldsymbol{y}_n \\ - \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \text{diag}\left(\left(\boldsymbol{D}^T\boldsymbol{D} \right)_{\boldsymbol{I}^t_{\boldsymbol{D},n}, \boldsymbol{I}^t_{\boldsymbol{D},n}} \right)^{-1} \text{diag}\left(\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}^T \boldsymbol{y}_n \right) \Bigg) \text{sign}\left(\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}^T \boldsymbol{y}_n \right)\end{aligned} \tag{7.6}$$

In the notation $\left(\boldsymbol{D}^T\boldsymbol{D} \right)_{\boldsymbol{I}^t_{\boldsymbol{D},n}, \boldsymbol{I}^t_{\boldsymbol{D},n}}$ the first index $\boldsymbol{I}^t_{\boldsymbol{D},n}$ tells which rows of $\left(\boldsymbol{D}^T\boldsymbol{D} \right)$ are used and the second index $\boldsymbol{I}^t_{\boldsymbol{D},n}$ tells which columns of $\left(\boldsymbol{D}^T\boldsymbol{D} \right)$ are used. Thus $\left(\boldsymbol{D}^T\boldsymbol{D} \right)_{\boldsymbol{I}^t_{\boldsymbol{D},n}, \boldsymbol{I}^t_{\boldsymbol{D},n}}$ gives the same square matrix as $\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}^T \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}$, but avoids computing it for each iteration of n. It is also seen that $\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}^T \boldsymbol{y}_n$ is computed on three separate occasions and should be computed only once and saved instead. This slightly increases the memory usage but decreases the computation time.

These changes are implemented in Listing 7.6.

Listing 7.6: Iteration 3 of the sequential optimization.

```
DtD = D_old.T@D_old
for n in range(N):
    DtY = D_old[:,I_D[:,n]].T @ Y[:,n]
    matproj = np.repeat(np.array([ D_old[:,I_D[:,n]] @ np.linalg.inv(DtD[I_D[:,n,None],
              I_D[:,n]]) @ DtY ]).T, S, axis=1)
    vecproj = D_old[:,I_D[:,n]] @ np.diag(np.diag( DtD[I_D[:,n,None], I_D[:,n]] )**-1*(DtY))
    signer = np.sign(DtY)
    D_new[:,I_D[:,n]] = D_new[:,I_D[:,n]]+(np.repeat(Y[:,n,None],S,axis=1)
                        -matproj+vecproj)*signer
```

7.3.1 Reevaluation

After the third iteration of optimization, the code is verified to run correctly as the resulting dictionary of the implementation is identical to the dictionary generated by the naïve implementation.

The execution time of the code in Listing 7.6 is found in Listing C.4 where it is seen that the total execution time is reduced from 16.02 s to 6.38 s.

7.4 Discussion

It would be possible to continue to attempt to optimize the code, thus yielding a smaller and smaller execution time. At this point there are no obvious sequential optimization steps to be made. Therefore the process of sequential optimization is not continued.

It should be noted that despite CPython utilizing a Global Interpreter Lock (GIL) more than one proces can be used simultaneously in the same program. This is due to the Numpy library which can bypass the GIL by running C code [25]. It has been decided not to limit the number of processes Numpy can use in this part as limiting it is not the typical use case.

When the implementation was optimized the focus was primarily on the execution time. However, in some applications the amount of memory used could be a critical factor. A test was performed in Appendix D to see if the memory usage had changed in the optimized code compared to the naïve implementation. Here the memory usage of the optimized code was found to be 3.856 MiB which is considered a small change from 3.344 MiB which was used by the naïve implementation. As the project is not performed on a platform where memory usage is an issue, an investigation into optimization of the memory usage is not performed.

8 | Parallel Optimization using CPU

The following chapter describes how to optimize the ITKrM algorithm in a parallel fashion using CPUs. This will be done by applying multiprocessing on iteration 3 of the sequential optimization found in Chapter 7. The following steps describes the chapters overall structure:

- A theoretical explanation of the multiprocessing Python library.
- An explanation of how the parallel optimized ITKrM algorithm will be evaluated.
- Implementation of the parallel optimized ITKrM algorithm.
- Evaluation of the parallel optimized ITKrM algorithm.

The third and the fourth bullets are repeated in each iteration of the optimization process. Furthermore, the influence of the inherent parallelism in the Numpy library, and the amount of overhead in the multiprocessing library are discussed.

8.1 The Multiprocessing Library

Four broad classifications are used when speaking of parallel computer architectures [26]. These four classifications are:

- Single Instruction, Single Data (SISD): Does not qualify for parallel processing [19].
- Single Instruction, Multiple Data (SIMD): The same instruction operates simultaneously on different parts of data. SIMD can be referred to as data parallel processing [19].
- Multiple Instruction, Single Data (MISD): Several processing units are performing different operations on the same data. This is rarely used [19].
- Multiple Instruction, Multiple Data (MIMD): Multiple processors performing operations on different parts of data. Once the processors are done computing, their results are joint to get the final result. Two subcategories of MIMD exist, Single Program, Multiple Data (SPMD) and Multiple Program, Multiple Data (MPMD). SPMD is when the multiple processors perform the same operation and it is similar to SIMD, but not as synchronized. MPMD is when the multiple processors perform different operations [19].

A number of tools for multiprocessing are included in the multiprocessing library in Python. Two of these are the map_async tool, which seems conceptually similar to SPMD, and the apply_async tool which can be thought of as MPMD [19]. It will be decided for each optimization iteration which tool will be used. As a principal rule, the map_async tool will be used when the problem is "embarrassingly parallel" i.e. the problem can be processed in parallel without any interaction, otherwise the apply_async tool will be used.

8.2 Techniques for Examining the Exploitation of Parallelism in the ITKrM Algorithm

To investigate how the parallelism of the ITKrM algorithm can be exploited, some techniques and tools can be used. These techniques and tools include Amdahl's and Gustafson-Barsis' laws, performance evaluation concerning execution time and memory usage, and examining the overhead when using multiprocessing.

8.2.1 Amdahl's and Gustafson-Barsis' laws

A part of applying multiprocessing to an algorithm is to select the number of processes which are used to process the problem. Depending on the relation between the parallel and the sequential part of the problem, the number of processes, it makes sense to use, changes. The aim of using more processes is that it should increase the computational performance of a specific problem. Two approaches to evaluate the speed-up potential of applying more processes to a problem are Amdahl's law and Gustafson-Barsis' law. Amdahl's law gives a conservative estimate of the upper limit of the speed-up potential and Gustafson-Barsis's law states that the speed-up potential goes to infinity as the number of processing units increases [19]. Whether it is appropriate to use Amdahl's or Gustafson-Barsis's law depends on the relation between the parallel and the sequential part of the problem, and can be divided into three categories:

- The sequential workload of the problem increases faster than the parallel workload as the problem size increases. In this case, it does not make sense to apply parallel computing.
- The relation between the sequential workload and the parallel workload of the problem is constant as the problem size increases. In this case, the speed-up potential has an upper limit and therefore it makes sense to consider Amdahl's law.
- The parallel workload of the problem increases faster than the sequential workload as the problem size increases. In this case, it makes sense to consider Gustafson-Barsis' law.

The three categories described above is under the assumption that the execution time is directly dependent on the problem size [19]. If it is found that Amdahl's law is applicable

the speed-up potential S_a, is found as

$$S_a = \frac{T_a(\alpha, :1)}{T_{a,seq}(\alpha)}, \tag{8.1}$$

where $T_a(\alpha, :1)$ is the execution time of the entire program using 1 thread and $T_{a,seq}(\alpha)$ is the sequential execution time. In order to find the appropriate number of processes Equation (8.2) is used [19].

$$S_{a,M} = \frac{M}{(M-1) \cdot \frac{1}{S_a} + 1} \tag{8.2}$$

In Equation (8.2) $\frac{1}{S_a}$ i.e. the sequential load, and M is the number of processes. Thus the speed-up potential can be found when using M processes. If it is found that Gustafson-Barsis' law is applicable it makes sense to use as many processes as available because the speed-up will keep increasing.

8.2.2 Performance evaluation

In Chapter 7 each optimization iteration was followed by a performance evaluation concerning execution time, and memory usage for the naïve implementation and 3rd sequential implementation. The line_profiler tool was used to evaluate the execution time of the program and the memory_profiler tool was used to evaluate the memory usage. These tools are however not applicable when map_async or apply_async is present in the code. Thus the parallel optimization iterations cannot be evaluated in the same way as the sequential optimization. The parallel optimization iterations will therefore only be evaluated on the total execution time and not on the execution time of the individual lines. The parallel optimization iterations will not be evaluated regarding memory usage. This is because it is not directly possible and because it is not prioritized to find a workaround to the conflict between the usage of multiprocessing and the memory_profiler tool. In section 6.1 it was stated that all tests should be made on the Phoenix 1 node in the Birdnest cluster. This is not the case for the parallel optimization on CPU and the reason is presented later in section 8.4.

8.2.3 Overhead when using Multiprocessing

Assume τ is the execution time of a certain embarrassingly parallel program, and M is the number of processes available. In the ideal case $\tau_{par} = \frac{\tau}{M}$, where τ_{par} is the execution time of the parallel program, when M processes are used to execute the program. This will not be the case since some overhead is probably involved in using multiprocessing. A procedure to measure the amount of overhead is presented in Appendix E.

8.3 First Iteration

In order to make a parallel implementation, it is important that the algorithm has inherent parallelism that can be exploited. When inspecting Equation (4.1) it is seen that there

is inherent parallelism in Equation (4.1a), as the computation of $\boldsymbol{I}^t_{\boldsymbol{D},n}$ is not dependent on previous or future n, and can be computed independently of each other. Similarly each atom in Equation (4.1b) is independent of other k. The computation of each atom was changed when optimizing sequentially to use matrix computations instead. The new computation of all atoms is shown for convenience in Equation (8.3).

$$\begin{aligned}\bar{\boldsymbol{D}} =& \frac{1}{N}\sum_n \Bigg(\boldsymbol{Y}_n + \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \left(\left(\boldsymbol{D}^T\boldsymbol{D} \right)_{\boldsymbol{I}^t_{\boldsymbol{D},n},\boldsymbol{I}^t_{\boldsymbol{D},n}} \right)^{-1} \boldsymbol{D}^T_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \boldsymbol{y}_n \\ & - \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \operatorname{diag}\left(\left(\boldsymbol{D}^T\boldsymbol{D} \right)_{\boldsymbol{I}^t_{\boldsymbol{D},n},\boldsymbol{I}^t_{\boldsymbol{D},n}} \right)^{-1} \operatorname{diag}\left(\boldsymbol{D}^T_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \boldsymbol{y}_n \right) \Bigg) \operatorname{sign}\left(\boldsymbol{D}^T_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \boldsymbol{y}_n \right)\end{aligned} \tag{8.3}$$

It contains a sum of all training examples where again all n are completely independent of each other. This means that the sum of all n can be split into small chunks that can be handled in parallel and added together at the end.

The normalization of each atom in the dictionary can also be done in parallel as all atoms are independent. The algorithm thus contains four parts with inherent parallelism that can be exploited. These are the computation of:

- The index set in Equation (4.1a).
- The atoms in Equation (4.1b).
- The sum for all n in Equation (8.3).
- The normalization of each atom in Equation (4.1c).

The computation of the sum for all n will be implemented using multiprocessing as Listing C.3 shows that the matrix and vector projection are the computationally heaviest, compared to computing the index set and normalizing the dictionary. Instead of exploiting the independence of each atom, it is chosen to use the matrix computation as the sequential optimization shows. This is done because the matrix computations are faster than looping through the vector computations stated in the mathematical formulation of ITKrM.

So what is done is similar to splitting the for-loop in Listing 7.4 into smaller chunks and allowing each available process to compute a chunk. The least amount of chunks it can be split into is the number of processes available, such that the processes get a chunk each to compute. This, however, means that the fastest total computation time depends on the slowest process. By splitting into smaller chunks it is possible for faster processes to handle more chunks while waiting for the slowest process, and therefore the overall computation time could decrease. It is assumed that an increase in the number of chunks increases overhead because of the management involved when processes finish and need

to start on a new chunk. So too small chunks is also a problem. It is decided that the data is split into 4 chunks per process available. The computational problem chosen to implement in parallel is a SPMD problem and is ideally suited for use with map_async. To implement this in parallel with multiprocessing, a pool of workers is created at first. This pool of workers is then to be given a function to run and an iterable of the data chunks. The function the workers should perform is the code in the for-loop in Listing 7.4. The data (training examples) is then split into a tuple that contains the data split into chunks, along with the corresponding index set split into similar chunks. This can be seen in Listing 8.1.

Listing 8.1: Code snippet of the first iteration of parallel optimization.

```
1  R = pool.map_async(_f, [(Y[:,n*N//(chunk_split*threads):(n+1)*N//(chunk_split*threads)], K, S,
       maxit, D_old, I_D[:,n*N//(chunk_split*threads):(n+1)*N//(chunk_split*threads)], n, DtD)
       for n in range((chunk_split*threads))]).get()
2  D_new = np.sum(R, axis=0)

4  def _f(d):
5      Y, K, S, maxitr, D_old, I_D, i, DtD = d
6      M, N = Y.shape
7      D_new = np.zeros((M, K))
8      for n in range(N):
9          DtY = D_old[:,I_D[:,n]].T @ Y[:,n]
10         matproj = np.repeat(np.array([ D_old[:,I_D[:,n]] @
                       np.linalg.inv(DtD[I_D[:,n,None], I_D[:,n]]) @ DtY ]).T, S, axis=1)
11         vecproj = D_old[:,I_D[:,n]] @
                       np.diag(np.diag( DtD[I_D[:,n,None], I_D[:,n]] )**-1*( DtY ))
12         signer = np.sign( DtY )
13         D_new[:,I_D[:,n]] = D_new[:,I_D[:,n]](np.repeat(np.array([Y[:,n]]).T, S, axis=1)
                               - matproj + vecproj)*signer
14     return D_new
```

8.3.1 Reevaluation

The code is verified as the resulting dictionary is identical to the generated dictionary created by the naïve implementation.

The first iteration of the parallel optimization has been tested in Appendix E against the three parameters: execution time, speed-up potential, and amount of overhead. The execution time when using nTrainingData $= 100$ and 4 processes, was found to be 1.75 s. From the execution time test in Appendix E, Figure 8.1 was made.

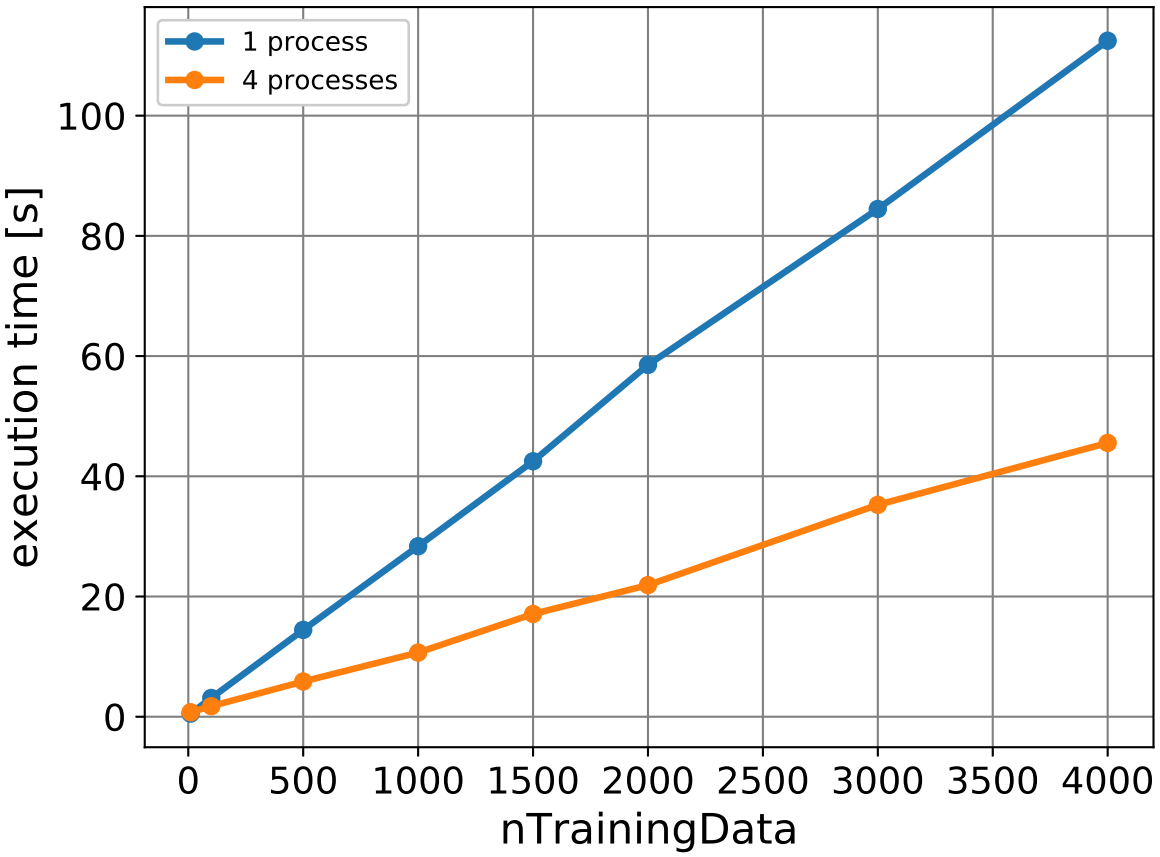


Figure 8.1: Execution using 1 and 4 processes.

It is seen in Figure 8.1 that as nTrainingData increases, so does the gap between the execution time when using 1 and 4 processes, in favour of the 4 processes. The execution time using 1 process is considered as a representative of the 3rd sequential optimized ITKrM implementation. From the speed-up potential test in Appendix E, Figure 8.2 was made.

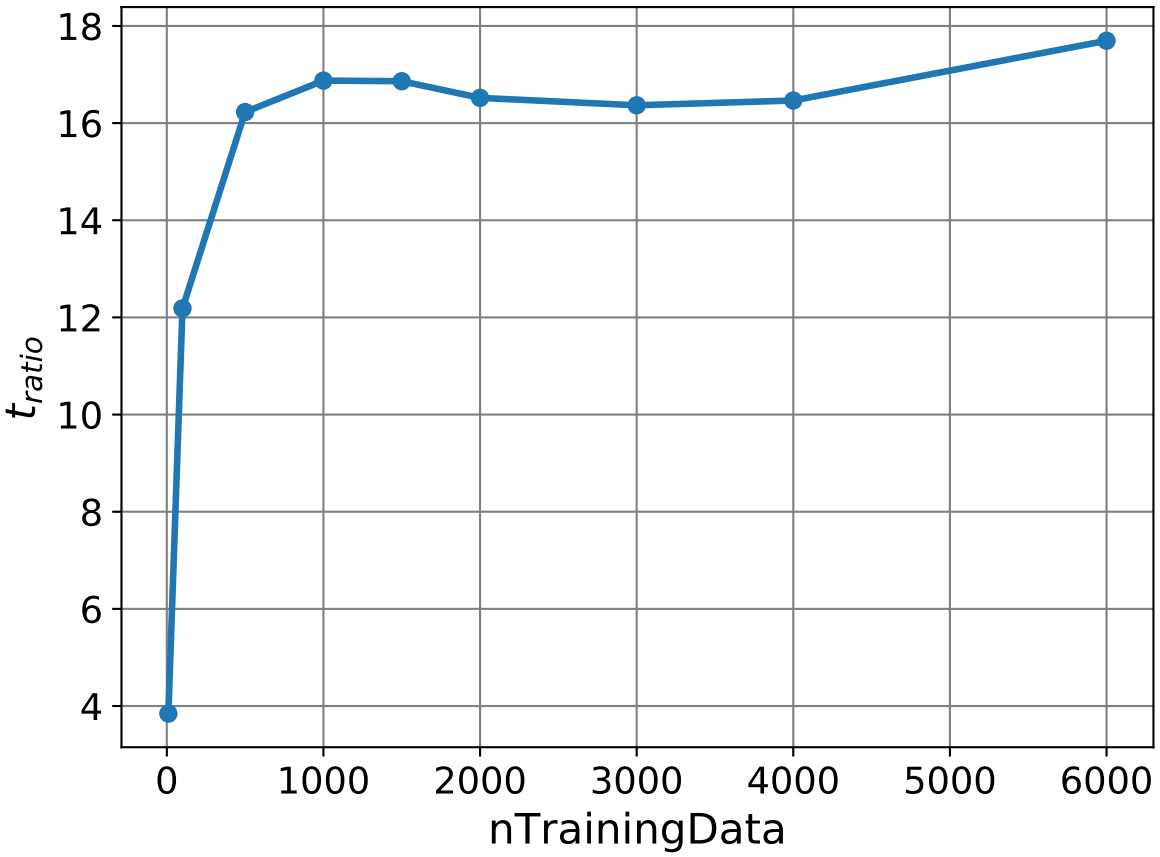


Figure 8.2: Graph of the ratio between the parallel and the sequential execution time.

It is seen in Figure 8.2 that the workload ratio between the parallel and the sequential part of the program t_{ratio} seems to converge. This indicates that a limit to the speed-up potential appears as the amount of training data increases. The upper limit to the speed-up potential can not be read directly from Figure 8.2, but it can be evaluated from Figure 8.2 that Amdahl's law can be used to determine this upper limit. This is done using the measured execution times in Table E.1 in Appendix E and inserting them in Equation (8.1). In Equation (8.4) the upper limit to the speed-up potential, when using 6000 training examples, is calculated.

$$S_a = \frac{T_a(\alpha, :1)}{T_{a,seq}(\alpha)} = \frac{175.61}{9.39} = 18.70 \tag{8.4}$$

Thus the ITKrM algorithm can increase its time performance with a factor of 18.70 when using multiprocessing after the first iteration of parallel optimization. By using Equation (8.2) and increasing M, the necessary number of processes to achieve a specific speed-up can be found and is presented in Table 8.1. From the overhead test in Appendix E, Figure 8.3 was made.

Table 8.1: Relation between number of processes and speed-up.

Speed-up	2	4	8	16
$\boldsymbol{M}$	3	5	14	176

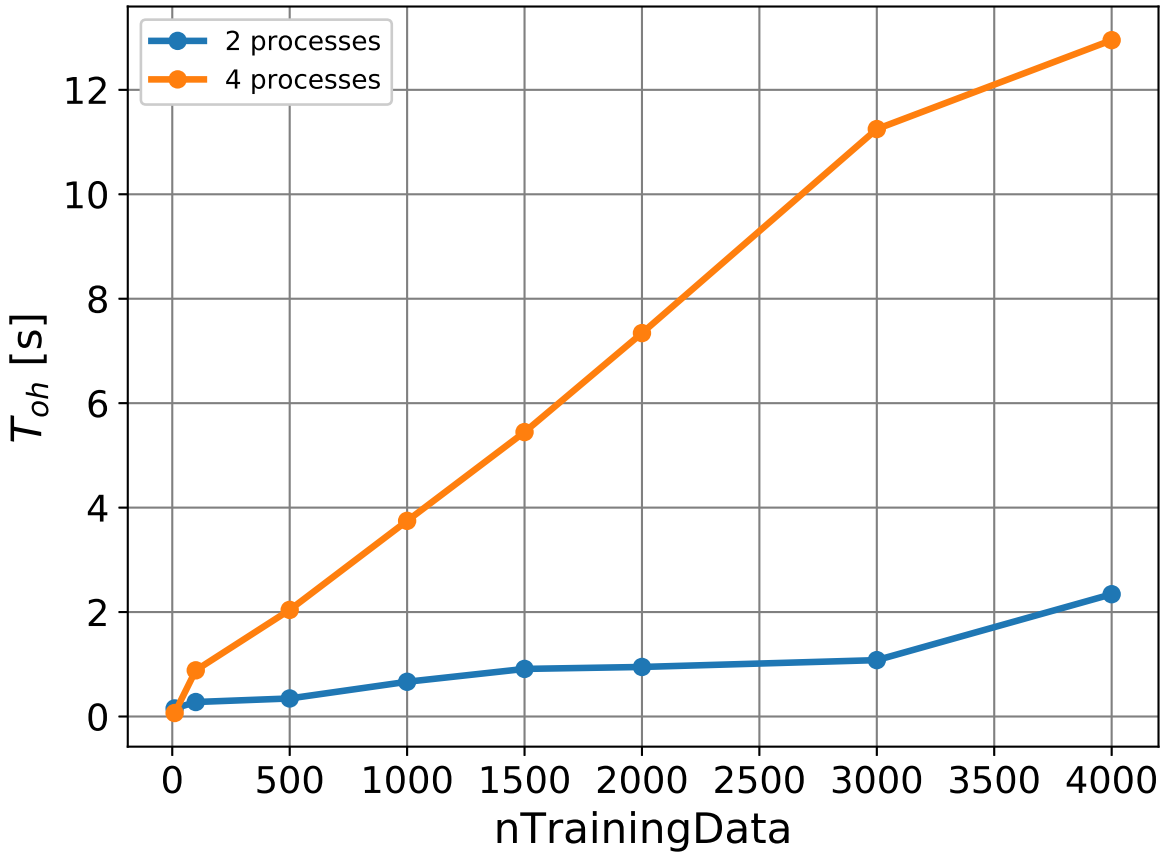


Figure 8.3: Amount of overhead using 2 and 4 processes.

It is seen in Figure 8.3 that the amount of overhead increases with the number of training data. The increase in overhead when using 4 processes has a linear tendency. The same tendency is seen when using 2 processes.

8.4 Discussion of the Parallel Optimization using CPU

A description has now been given of a parallel CPU implementation of the ITKrM algorithm. The implementation has been analyzed and evaluated relative to Amdahl's law together with a performance and overhead evaluation. From Equation (8.4) it was found that the use of multiprocessing should be able to reach a speed-up of 18.70 times. In Appendix F an attempt to verify this was made, where both the Phoenix 1 node in the

Birdnest cluster and a reference PC were used. The specifications of the reference PC can be seen in Appendix F. In Figure F.1 it is shown that the ITKrM algorithm does not decrease its execution time as expected, when increasing the number of processes. The purpose of using the Phoenix node was that it would be possible test all implementations on this unit and also test for different numbers of processes. This purpose is corrupted by the speed-up tendency found in Figure F.1. An identical test was made on a reference PC in order to verify if the problem was the result in Equation (8.4) or the Phoenix node. The result of the test on the reference PC is shown in Figure 8.4.

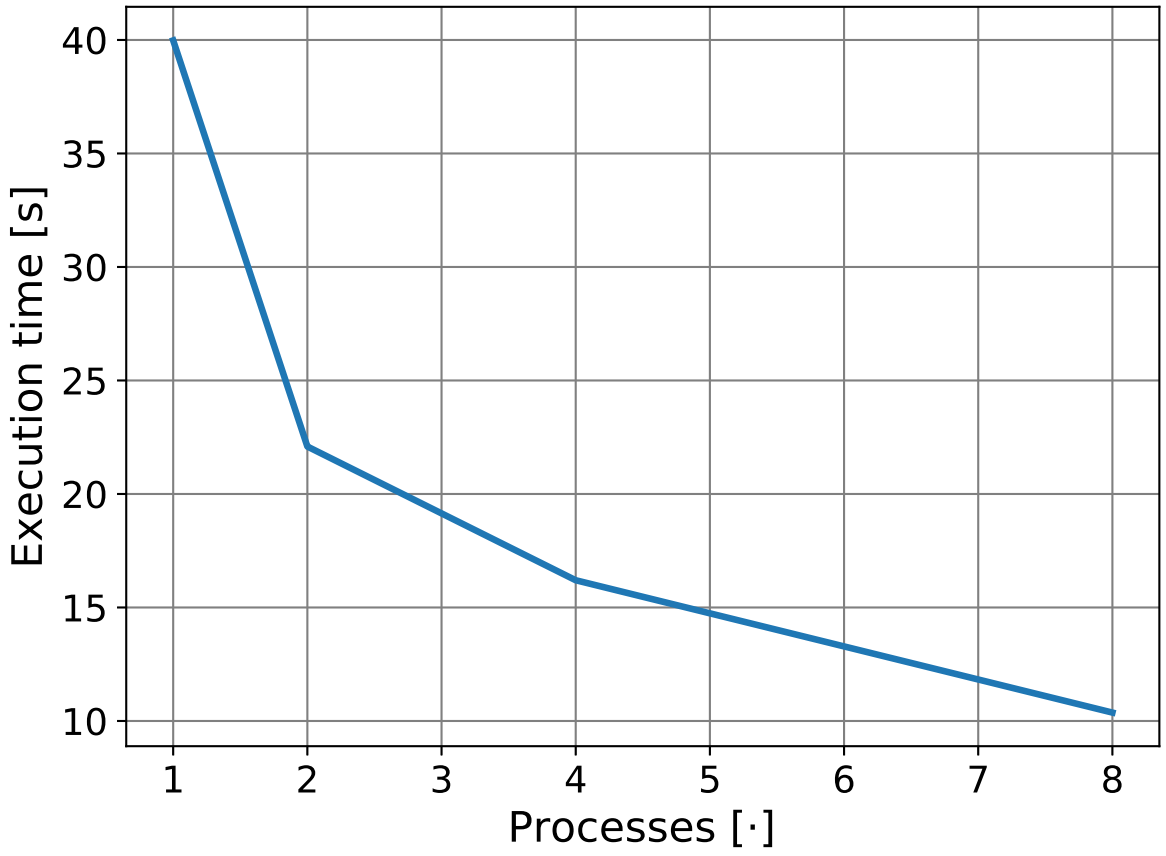


Figure 8.4: Shows the execution for increasing number of processes on reference PC.

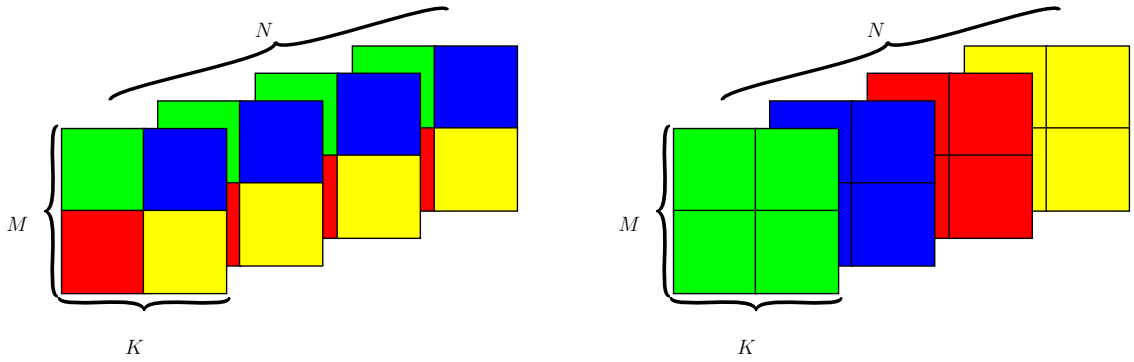
Figure 8.4 shows that the execution time is approximately halved every time the number of processes is doubled, which is the tendency that was expected. Thus it is concluded the disagreement between the result in Equation (8.4) and Figure F.1 is because of a problem in the Phoenix node. This defeats the purpose of using the Phoenix node for testing the multiprocessing on CPU. This is the reason why the test for execution time for multiprocessing on CPU Figure 8.1 is made on a different unit than the Phoenix 1 node.

9 | Parallel Optimization using GPU

This chapter describes the implementation of the ITKrM algorithm on a GPU. As discussed in section 5.1 the GPU is highly efficient for doing numerical calculations, and as such is viable to be used for training a dictionary. The ITKrM algorithm has features which make it reasonable to consider implementing it on a GPU. Based on the mathematical description of the algorithm in section 4.1 the following parallel computations of the algorithm are well suited for parallel processing.

- The index set in Equation (4.1a).
- The atoms in Equation (4.1b).
- The sum for all n in Equation (8.3).
- The normalization of each atom in Equation (4.1c).
- Each entry of the matrix operations.

Figure 9.1 illustrates two different ways of computing a dictionary in the ITKrM algorithm. Figure 9.1a illustrates the computations of each entry computed in parallel while Figure 9.1b illustrates that the dictionary is computed as a sum of several sub-dictionaries which are computed in parallel.



(a) Computing each matrix entry in parallel. **(b)** Computing each sub-dictionary in parallel.

Figure 9.1: Illustration of how parallel processing can be used to compute a dictionary. Each color represent an individual thread.

In order to implement the ITKrM algorithm on a GPU some research on how to interact with the GPU must be done.

9.1 Interacting with a GPU for Numerical Calculations

The tools to interact with GPUs have in recent years been under development. The GPU manufacturer NVIDIA has been a driving force in this development by producing Compute Unified Device Architecture (CUDA)-toolkit, which is an Application Programming Interface (API) that eases the interacting with a GPU. It has the downside of only working on NVIDIA GPUs [27]. An open source alternative to this is Open Computing Language (OpenCL) which has many similarities to CUDA, but works for all GPUs [28]. In this project it is chosen to work with CUDA to interface with a GPU, meaning only NVIDIA GPUs are suitable.

CUDA is an API where the programmer has to write C code to connect and handle calculations on the GPU. This is done through defining kernels or device functions and handling data transfer to and from the GPU. Kernels are functions that can be called from the host (CPU), whereas device functions are functions only the GPU may call. This project tries to work with the higher level programming language Python. One way to achieve this is through the use of a module for Python, called PyCUDA. It allows for access to a GPU through coding in Python. It still requires the user to write C inside Python, so it is simply CUDA in Python. In this project, it is of interest to have modules that instead offers some higher level programming. Such modules are readily available, and some fairly know modules are: Numba, Pyculib, ArrayFire, and CuPy. To better choose among them some background knowledge of each module is necessary.

Numba is primarily known for its Just In Time (JIT)-compiling, but also offers support for GPUs. Numba allows writing code in Python and then through the use of a decorator, Numba knows that function must be converted into GPU compatible code. There is, of course, some restrictions when writing the kernels or device functions, and the functions have to be called with thread and block sizes defined [29]. An example of how to code with Numba is shown in Listing 9.1.

Listing 9.1: Snippet of Numba coding for CUDA

```
@cuda.jit
def matmul(A, B, C):
    #Perform square matrix multiplication of C = A * B
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]: # To ensure we do not run outside the arrays
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp

matmul[blockspergrid, threadsperblock](A, B, C)
```

Pyculib serves to give easier access to NVIDIA's libraries e.g. linear algebra. It provides some easier interfacing with these library packs compared to CUDA written directly in C.

It requires Numba, so it is more of an addon with some premade GPU enabled functions [30].

ArrayFire allows for code written in Python to be converted for GPU. This is much like Numba, but compared to Numba and the other listed modules, ArrayFire has native support for both CUDA and OpenCL, through the use of JIT-compiling. A simple function call in the start of the function can define which backend platform should be used. Furthermore, ArrayFire has submodules that provides already made functionality for quite some areas, e.g. linear algebra [31]. An example of how to code with ArrayFire is shown in Listing 9.2.

Listing 9.2: Snippet of ArrayFire coding for CUDA

```
# Monte Carlo estimation of pi
def calc_pi_device(samples):
    af.set_backend(Target) #e.g CUDA or OpenCL
    # Generate uniformly distributed random numbers
    x = af.randu(samples)
    y = af.randu(samples)
    # The following line generates a single kernel
    within_unit_circle = (x * x + y * y) < 1
    return 4 * af.count(within_unit_circle) / samples
```

The CuPy module is a module with functionalities similar to Numpy, but the functions are made to run on a GPU. Most functionalities Numpy has is also available in CuPy, and nearly a whole program can be made to run on a GPU, simply by changing from Numpy.* to CuPy.*, as they share names for nearly all their functionality. This makes it a module that allows for easy conversion from CPU to GPU while using a high-level syntax [32]. An example of how to code with CuPy is shown in Listing 9.3.

Listing 9.3: Snippet of CuPy coding for CUDA

```
import numpy as np
import cupy as cp
#Matrix multiplication
GPU_A = cp.asarray(np.random.rand(10, 10))
GPU_B = cp.asarray(np.random.rand(10, 10))
GPU_Result = cp.dot(A, B)
CPU_Result = cp.asnumpy(GPU_result) # To return the value to the CPU
```

It should be noted that there exists an incompatibility between CuPy and the compute capability of the GPUs in the Birdnest cluster. The CuPy module requires a compute capability of the GPUs to be at least 3.0, however, the compute capability of the Phoenix nodes are only 2.0. It is therefore not desirable to use the GPU compute nodes in the Birdnest cluster for test and implementation.

9.2 Initial Test of the GPU

To get an initial grasp of what kind of speed up it might yield, and in which areas it might be worth applying a GPU, two test were made in Appendix G. In Figure 9.2 the time it takes to invert a matrix on CPU versus the time it takes on GPU is shown. This is relevant as the heavy part of the ITKrM algorithm is a projection that requires a matrix inversion.

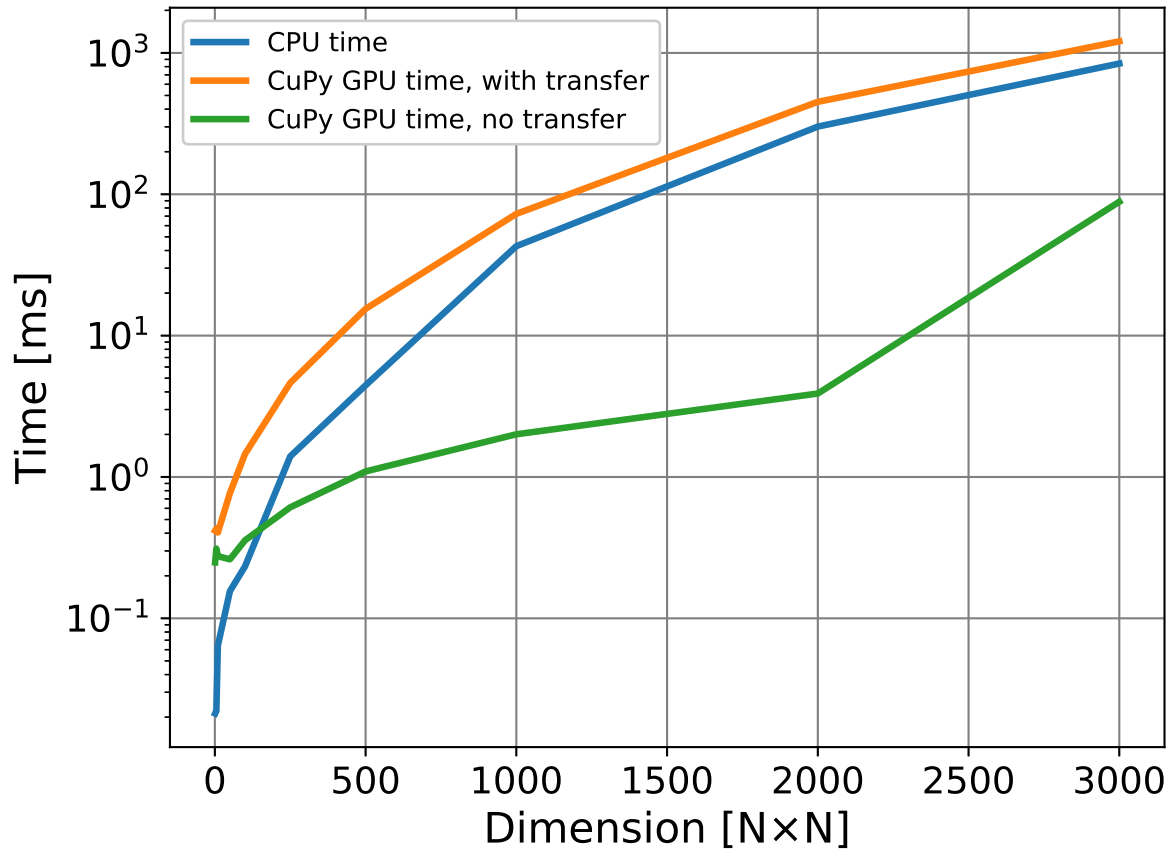


Figure 9.2: Shows the time it takes to do a matrix inversion of a size N by N matrix.

What can be seen from Figure 9.2 is that even though the GPU is faster, the time it takes to send information to the GPU makes it appear slow. This means that effort should be put into making the GPU do a lot of computations on the data that is sent. The second test in Appendix G evaluates the execution time of multiplying two N×N matrices on a CPU and a GPU. The second test is made because matrix multiplication is a typical method of evaluating a processor's computational performance. The result of the test is seen in Figure 9.3.

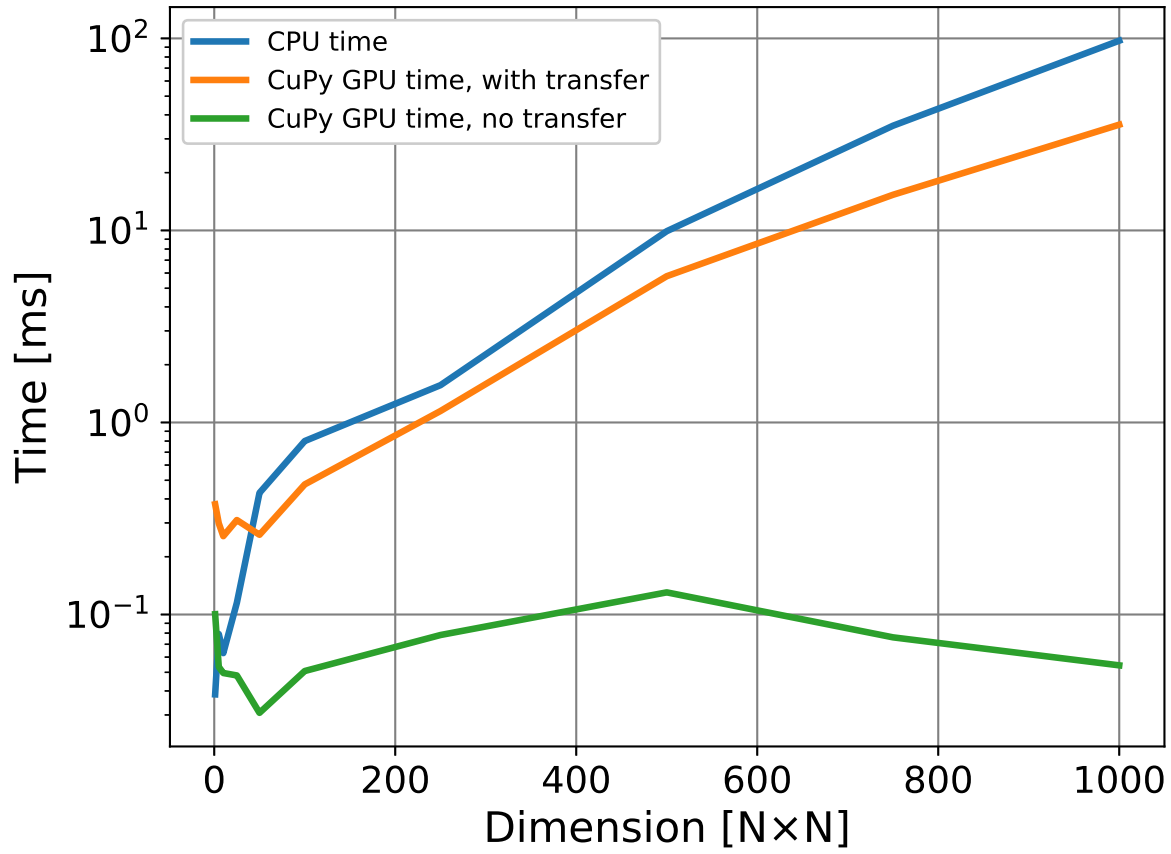


Figure 9.3: Shows the execution time of square matrix multiplication on CPU and GPU.

What is seen in Figure 9.3 is that when taking the data transfer from CPU to GPU and back into account, the execution time of the CPU and the GPU with transfer follows quite closely. It is important to notice that the CPU is the fastest of the two until the dimensions of the matrices reach approximately 700×700. From here the GPU with transfer is faster than the CPU, a tendency that might continue if the dimensions were further increased. What can also be seen from Figure 9.3 is that as the dimension of the matrices passes approximately 100×100, the GPU without transfer becomes the fastest of the three. The results from Figure 9.3 supports the statement that when using a GPU the data transfer from CPU to GPU and back should be considered.

It is chosen that for the first/naïve implementation of ITKrM, CuPy will be used, where the 3rd iteration from the sequential optimization will be used as the base.

9.3 Performance Evaluation

The tools for implementing a naïve ITKrM algorithm on a GPU has now been presented. The naïve GPU implementation will not have a focus on execution time, but is made

solely to have a functioning and validated implementation of the ITKrM algorithm on a GPU. The execution time of the naïve implementation will still be evaluated. The naïve implementation on the GPU will be verified by comparing the dictionary found in the naïve GPU implementation with the verified dictionary.

The memory usage will not be evaluated on any of the GPU implementations since it can not be verified if the memory_profiler will return the memory used on the CPU or the memory used on the GPU. The line_profiler tool will be used to examine the execution time of the GPU implementations in order to give an overview of the heaviest tasks of the program. As seen in section 5.1 the transfer of data between the CPU and the GPU is something the developer should consider. The developer should aim to transfer data from CPU to GPU and back, as rarely as possible. The nvprof tool [33] is used to get an overview of the influence of data transfer w.r.t. execution time.

As seen in Appendix H the GPU implementations are tested on a different device than the sequential- and the CPU parallel- implementations. This complicates the execution time comparison between the GPU implementations and the other implementations. To make the comparison, the execution time of the GPU implementations will be measured for increasing problem size and compared to the 3rd sequential implementation, where both tests are made on the same device.

9.4 Naïve GPU implementation of ITKrM

The implementation of ITKrM is done by replacing all the calls to Numpy with a call to CuPy. For this project that entails changing function calls from np.* to cp.*. Furthermore, in the start of the ITKrM algorithm the static values are transferred to reduce the amount of data transfer to and from the GPU in the loop over N. The naïve implementation of ITKrM for GPU can be seen in Listing 9.4.

Listing 9.4: The naïve implementation of ITKrM for GPU

```
#Algorithm
GPU_D_old = cp.asarray(D_init)
GPU_Y = cp.asarray(data)
GPU_M = int(cp.asarray(M))
GPU_N = int(cp.asarray(N))
GPU_S = int(cp.asarray(S))
GPU_maxitr = int(cp.asarray(maxitr))
GPU_I_D = cp.zeros((S,N),dtype=cp.int32)

for i in range(GPU_maxitr):
    # Finding index set
    for n in range(GPU_N):
        GPU_I_D[:,n] = max_atoms(GPU_D_old,GPU_Y[:,n],GPU_S)

    GPU_D_new = cp.zeros((M,K))
    GPU_DtD = GPU_D_old.T @ GPU_D_old

    # Updating the dictionary
```

```
    for n in range(GPU_N):
        # Precalculating DtY
        GPU_DtY = GPU_D_old[:, GPU_I_D[:, n]].T @ GPU_Y[:, n]
        GPU_matproj = cp.repeat((GPU_D_old[:, GPU_I_D[:, n]] @ cp.linalg.inv(GPU_DtD[GPU_I_D
                      [:, n, None], GPU_I_D[:, n]]) @
                      GPU_DtY)[:, None], GPU_S,axis=1)
        GPU_vecproj = GPU_D_old[:, GPU_I_D[:, n]] @
                      cp.diag(cp.diag( GPU_DtD[GPU_I_D[:, n, None],
                      GPU_I_D[:, n]] )**-1*( GPU_DtY ))
        GPU_signer = cp.sign( GPU_DtY )
        GPU_D_new[:, GPU_I_D[:, n]] = GPU_D_new[:, GPU_I_D[:, n]] +
                                      (cp.repeat(GPU_Y[:, n, None], S, axis=1)
                                      - GPU_matproj + GPU_vecproj)*GPU_signer

    # Normalize the dictionary
    GPU_scale = cp.sum(GPU_D_new*GPU_D_new, axis=0)
    GPU_iszero = cp.where(GPU_scale < 0.00001)[0]
    GPU_D_new[:, GPU_iszero] = np.random.randn(GPU_M, len(GPU_iszero))
    GPU_D_new = normalize_mat_col(GPU_D_new)
    GPU_D_old = 1*GPU_D_new
return cp.asnumpy(GPU_D_old)
```

Now that a funtional implementation on GPU for the ITKrM algoritm has been made, and evaluation of said implementation can be made.

9.4.1 Evaluation of the Naïve GPU Implementation

The naïve GPU implementation has been validated to function correctly, see Appendix H. The execution time of the naïve GPU implementation for different size of training data is shown in Figure 9.4.

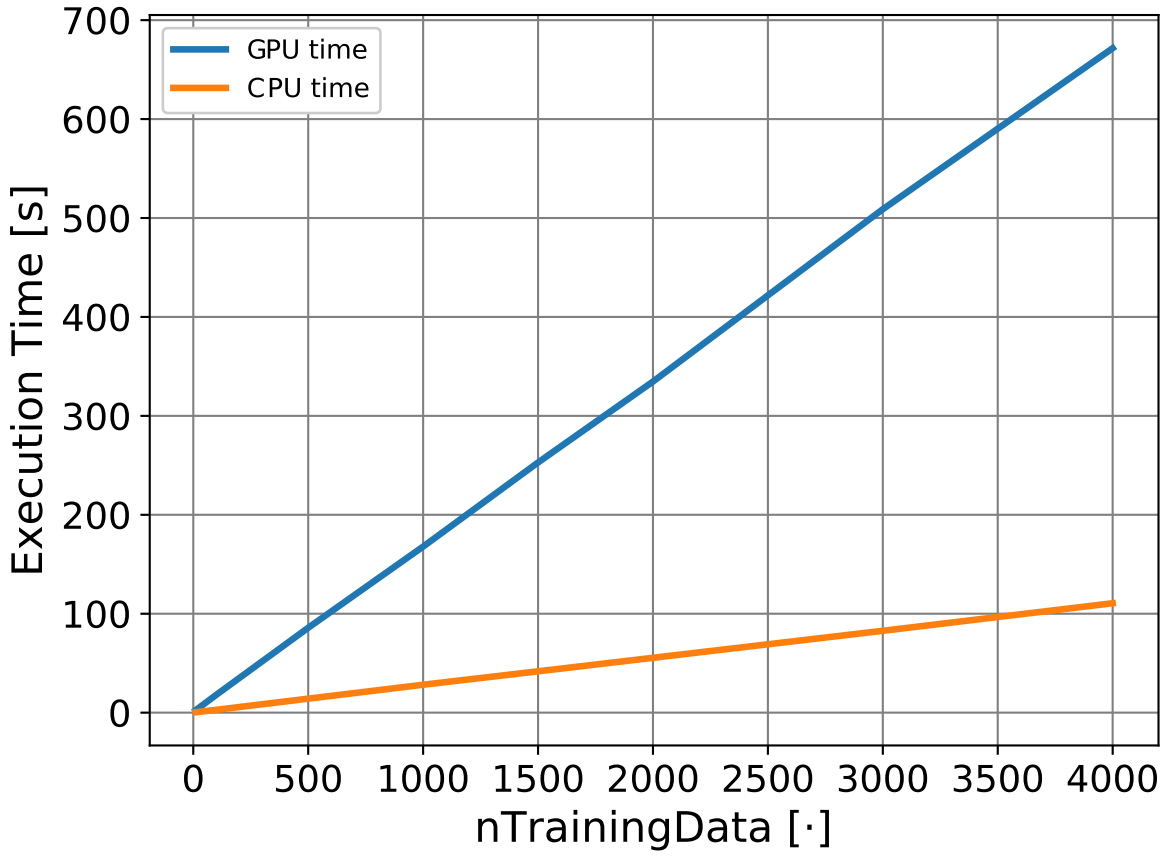


Figure 9.4: Shows the execution times of the ITKrM algorithm. CPU vs. GPU.

What is seen in Figure 9.4 is that the GPU implementation is slow. When inspecting the code in Listing 9.4 it can be seen that there is a for-loop over the training data, line 17. These training data can be calculated independent of each other, but the way the code is implemented in Listing 9.4, they are run one after each other. This does most likely not utilize all of the GPUs computing power.

The line profiling showed much the same percentage numbers for which lines takes up most of the execution time, see Listing H.1. Lastly when inspecting the results of the nvprof test in Appendix H, it can be seen that the transfer of data takes a very low percentage of the total execution time, see line 33 and 39 in Listing H.2. Upon further inspection, there are two commands which take around 10% of the GPU activity, cupy_copy and cupy_take, line 7 and 8 in Listing H.2. These commands might also have something to do with data transfer, but it has not been possible to determine whether it is simply internal transfer on the GPU, or if it is transfer between the GPU and CPU.

It would be of interest to try and make the GPU run all the training data at once, or at least in some chunks, similar to how the parallel implementation was done for CPU. This

is because the current implementation only utilized the parallelizability of the GPU in each function call, and not utilizing the inherent parallelism of the ITKrM algorithm. It could also be of interest to see if only running a select amount of the calculations on the GPU, could yield some better results on the speed, as maybe not all the functions called on the GPU are faster. Furthermore, it could be discussed that the size of the problem used for testing, might not be big enough for the GPU to be the fastest. On the more positive side, the implementation was extremely easy to make when using CuPy, as it was not much harder than if written with the Numpy module.

9.5 Custom CUDA Kernel Implementation

As the naïve GPU implementation was slower than the 3rd sequential implementation, it would be interesting to try and implement it in a similar manner to that of the parallel implementation on the CPU. On the CPU, it was the summation for all n that was split into chunks and executed in parallel. Each processor handled a smaller chunk of data and returned the result, before summing it to obtain the updated dictionary. The GPU has many more processing cores to utilize than the CPU and thus has a very high speed-up potential as showcased in the speed-up graph in Figure 8.4.

There is no general purpose CUDA library that can execute the ITKrM algorithm on the GPU in this manner. Most of these libraries handle the computation of each linear algebra function in parallel and not multiple series of functions in parallel on different data. This means that a custom CUDA kernel has to be coded that implements the specific functionality to compute an updated dictionary for each training example in parallel.

This can be written in C using the CUDA toolkit from NVIDIA or it can be written in Python using the Numba library. The Numba library handles all memory allocation and data transfer between CPU and GPU and hides data structures, while it needs to be done manually if using C. Numba will therefore be used to write the CUDA kernel.

9.5.1 Kernel Initialization

A kernel function in Numba is initialized as shown in Listing 9.5.

Listing 9.5: Kernel initialization in Numba.

```
1 from numba import cuda

3 @cuda.jit
4 def kernel(input_array, output_array):
5     # Kernel code
```

The kernel gets some arrays as parameters which are copied from the host (CPU) to the device (GPU). The GPU then performs all its computations and once the kernel completes, the arrays are copied back from the device to the host and replaces the original host arrays.

This is a part of the automatic memory handling, but Numba has functions that allow for more control in that regard if necessary [34]. As a result, there is no return call in the kernel function and care should be taken to not overwrite data in input arrays.

It is chosen to unroll the matrices, as this can simplify some of the control needed. So the matrices need to be transformed into long arrays. This will be done by appending each row of the matrices, such that each row will lie next to each other in the arrays. Accessing element $a_{i,j}$ of a matrix $\boldsymbol{A} \in \mathbb{R}^{M \times K}$ after this transformation is done by $A[j+i \cdot K]$. If there are N instances of $\boldsymbol{A}$ the element $a_{i,j}$ of the n-th instance is accessed by $A[j+i \cdot K+n \cdot M \cdot K]$.

All processors execute the same kernel in parallel and all have access to the data given to the kernel. Thus each processor needs a way to know which parts of the data to operate on. Otherwise, all processors will just be doing the same calculations on the same data, which does not provide any benefits w.r.t. execution time.

The CUDA extension to the C language provides a function call that returns the ID of the processor and is accessible via Numba. Each processor has a thread ID and a block ID. It is possible to set the number of threads per block and the number of blocks per grid when calling the kernel. The number of threads per block cannot exceed 1024 [35, p. 7-10]. The number of blocks per grid depends on the size of the problem. For example, to do a matrix multiplication of two matrices $\in \mathbb{R}^{64 \times 64}$ entirely in parallel, there needs to be $64 \cdot 64$ threads running. Each thread will calculate a single entry of the resulting matrix. If the number of threads per block is set to 32 then the number of blocks per grid needs to be $\frac{64 \cdot 64}{32} = 128$. If these numbers are set too low some entries will not be computed, or the workload has to be distributed such that each processor needs to calculate more than one entry. Listing 9.6 shows the way to call a kernel and how to access the processor ID.

Listing 9.6: Kernel call and determining processor ID.

```
1  from numba import cuda
2
3  @cuda.jit
4  def kernel(input_array, output_array):
5      n = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
6      # n is a unique number that represents the processor ID.
7      # Kernel code
8
9  a = np.array([1,2])
10 b = np.array([2,1])
11 threadsPerBlock = 32
12 blocksPerGrid = 1
13 kernel[threadsPerBlock, blocksPerGrid](a, b)
```

With the kernel initialization and how to call it described, the different parts of math in the ITKrM algorithm can be implemented. These parts are the vector projection, the matrix projection and the summation of $\boldsymbol{D}$ for all n.

9.5.2 ITKrM on CUDA Kernel - Vector Projection

As each processor needs to compute the vector projection on a different training example, it needs to perform all matrix operations sequentially for each entry of the matrix. A simple way to go from matrix operations to a sequential implementation is to write up the entry-wise calculations necessary and then make an implementation based on that. This is done by first inspecting the computation of the vector projection in Equation (9.1).

$$\text{vecproj} = \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \text{diag}\left(\boldsymbol{D}^T_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}\right)^{-1} \text{diag}\left(\boldsymbol{D}^T_{\boldsymbol{I}^t_{\boldsymbol{D},n}} \boldsymbol{y}_n\right) \tag{9.1}$$

Multiplication with the diagonal matrices is just a multiplication of each diagonal element on the corresponding column of $\boldsymbol{D}_{\boldsymbol{I}^t_{\boldsymbol{D},n}}$. The vector projection $\boldsymbol{V}$ is then calculated entry-wise in Equation (9.2) remembering that $\boldsymbol{d}_{i_s}$ is the i_s-th column of $\boldsymbol{D}$ and i_s is the s-th element in $\boldsymbol{I}_{\boldsymbol{D},n}$. Element d_{m,i_s} is then the element in the m-th row and i_s-th column of $\boldsymbol{D}$.

$$v_{m,s} = d_{m,i_s} \sum_{j=0}^{M-1} (d_{j,i_s} \cdot d_{j,i_s})^{-1} \sum_{j=0}^{M-1} (d_{j,i_s} \cdot y_j) \qquad s = 0, 1, ..., S-1 \quad , \quad m = 0, 1, ..., M-1 \tag{9.2}$$

The vector projection $\boldsymbol{V}$ is computed for n different training examples and is saved one after the other in a single array. The computation of the vector projection is then implemented in a kernel function and is seen in Listing 9.7

Listing 9.7: Kernel Vector Projection.

```
@cuda.jit
def k_matvecproj(M, N, K, S, D, DtD, I_D, Y, vecproj):
    n = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    if (n < N):
        ### Vector projection
        for i in range(S):
            DtY = 0
            for j in range(M):
                DtY += D[I_D[n + i * N] + j * K] * Y[n + j * N]
            # Calculate D/diag(DtD)*DtY
            for m in range(M):
                vecproj[i + m*S + n*M*S] = D[I_D[n + i*N] + m*K] * DtY / DtD[ I_D[n + i*N] +
                                           I_D[n + i*N]*K ]
```

9.5.3 ITKrM on CUDA Kernel - Matrix Projection

The matrix projection is similarly in need of being calculated sequentially on each processor. It does, however, come with the added problem of a matrix inversion of a matrix that is not a diagonal matrix. A stable and computationally efficient algorithm for matrix inversion involves the LU decomposition [36]. The LU decomposition needs a pivot matrix that swaps the rows of the matrix so the largest absolute value of each column is on the diagonal [36]. This is done for numerical stability. However, as the dictionary

$\boldsymbol{D}$ is normalized i.e. the norm of each column equals 1 and the matrix to be inverted is $\boldsymbol{D}^T\boldsymbol{D}$, the diagonal consists of 1s and off-diagonal elements that are smaller than one. This means that the matrix is already in the correct permutation and the pivot matrix is irrelevant. For an in-depth description of the LU decomposition and how it can be used to solve a system of linear equations, see Appendix I.

The matrix projection is computed as shown in Equation (9.3).

$$\text{matproj} = \boldsymbol{D}_{\boldsymbol{I}^t_{D,n}} \left(\left(\boldsymbol{D}^T\boldsymbol{D} \right)_{\boldsymbol{I}^t_{D,n}, \boldsymbol{I}^t_{D,n}} \right)^{-1} \boldsymbol{D}^T_{\boldsymbol{I}^t_{D,n}} \boldsymbol{y}_n \tag{9.3}$$

As $\boldsymbol{D}^T_{\boldsymbol{I}^t_{D,n}} \boldsymbol{y}_n$ results in a vector and $\left(\boldsymbol{D}^T\boldsymbol{D} \right)_{\boldsymbol{I}^t_{D,n}, \boldsymbol{I}^t_{D,n}}$ is a square matrix, the partial result after the first matrix $\boldsymbol{D}_{\boldsymbol{I}^t_{D,n}}$ can be viewed as the solution to Equation (9.4).

$$\left(\boldsymbol{D}^T\boldsymbol{D} \right)_{\boldsymbol{I}^t_{D,n}, \boldsymbol{I}^t_{D,n}} \boldsymbol{x} = \boldsymbol{D}^T_{\boldsymbol{I}^t_{D,n}} \boldsymbol{y}_n \tag{9.4}$$

This can be solved using the LU decomposition and the algorithm shown in Listing I.1 and Listing I.2. The matrix projection can then be computed as shown in Equation (9.5).

$$\text{matproj} = \boldsymbol{D}_{\boldsymbol{I}^t_{D,n}} \boldsymbol{x} \tag{9.5}$$

The solution $\boldsymbol{x}$ is already computed entry-wise in Appendix I and the matrix projection $\boldsymbol{q}$ can then be computed entry-wise as shown in Equation (9.6).

$$q_m = \sum_{s=0}^{S-1} d_{m,i_s} \cdot x_s \qquad m = 0, 1, ..., M-1 \tag{9.6}$$

The matrix projection can then be implemented in the same kernel function and similarly to the vector projection, the matrix projection for each training example is saved one after the other in an array.

Listing 9.8: Kernel Matrix Projection.

```
1  @cuda.jit
2  def k_matvecproj(M, N, K, S, D, DtD, I_D, DtY, Y, vecproj, L, U, z, x, matproj):
3      n = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
4      if (n < N):
5          ### Vector projection
6          # ...

14         ### LU decomposition
15         for i in range(0, S):
16             for j in range(i, S):
17                 L[i + j*S + n*S*S] = DtD[I_D[n + i*N] + I_D[n + j*N]*K]
18                 for k in range(0, i):
19                     L[i + j*S + n*S*S] -= L[k + j*S + n*S*S] * U[i + k*S + n*S*S]

21             U[i + i*S + n*S*S] = 1
```

```
22              for j in range(i+1, S):
23                  U[j + i*S + n*S*S] = DtD[I_D[n + j*N] + I_D[n + i*N]*K] / L[i + i*S + n*S*S]
24                  for k in range(0, i):
25                      U[j + i*S + n*S*S] -= (L[k + i*S + n*S*S] * U[j + k*S + n*S*S])
                                              / L[i + i*S + n*S*S]

27          ### Solve system
28          for i in range(0, S):
29              z[i + n*S] = DtY[i + n*S] / L[i + i*S + n*S*S]
30              for j in range(0, i):
31                  z[i + n*S] -= (L[j + i*S + n*S*S] * z[j + n*S]) / L[i + i*S + n*S*S]
32          for i in range(S-1, 0-1, -1):
33              x[i + n*S] = z[i + n*S]
34              for j in range(i+1, S):
35                  x[i + n*S] -= U[j + i*S + n*S*S] * x[j + n*S]

37          ### D@x
38          for m in range(0, M):
39             matproj[m + n*M] = 0
40             for s in range(0, S):
41                 matproj[m + n*M] += D[I_D[n + s*N] + m*K] * x[s + n*S]
```

9.5.4 Update Dictionary using Results from CUDA Kernel

The kernel function now computes the vector- and matrix projection for all n in parallel and the dictionary can now be updated by calculating the sum for all n in Equation (4.1b). The rest will be done on the CPU.

The vector- and matrix projections that have been computed are both arrays and need to be put back into their original shape. This is done by using the reshape function in Python. The ITKrM algorithm can then be implemented to use the custom CUDA kernel by replacing lines 2-7 in Listing 7.6 with Listing 9.9.

Listing 9.9: Python code for updating the dictionary after the GPU has computed the matrix and vector projections.

```
1  # Create arrays for GPU to use
2  d_D = D.reshape(-1)*1
3  d_DtD = DtD.reshape(-1)*1
4  d_I_D = I_D.reshape(-1)*1
5  d_DtY = np.zeros(N*S)
6  d_Y = Y.reshape(-1)*1
7  d_vecproj = np.zeros(N*M*S)
8  L = np.zeros(N*S*S)
9  U = np.zeros(N*S*S)
10 z = np.zeros(N*S)
11 x = np.zeros(N*S)
12 d_matproj = np.zeros(M*N)
13 # Compute matrix and vector projections in parallel
14 k_matvecproj[TpB, BpG](M, N, K, S, d_D, d_DtD, d_I_D, d_DtY, d_Y, d_vecproj, L, U, z, x,
                          d_matproj)
15 # Update dictionary on CPU
16 for n in range(N):
```

```
17      DtY = D[:,I_D[:,n]].T @ Y[:,n]
18      matproj = np.repeat(d_matproj.reshape(N, M)[n,:,None], S, axis=1)
19      vecproj = d_vecproj.reshape(N,M,S)[n,:,:]
20      signer = np.sign( DtY )
21      D_new[:,I_D[:,n]] = D_new[:,I_D[:,n]] + (np.repeat(Y[:,n,None], S, axis=1) -
                            matproj + vecproj)*signer
```

The custom CUDA kernel is now implemented and ready to be tested.

9.5.5 Evaluation of the Custom CUDA Kernel Implementation

All the tests that were made in order to evaluate the custom CUDA kernel implementation are found in Appendix J. The implementation is tested for correctness in subsection J.5.1 by seeing that it produces an identical dictionary as the dictionary computed by the sequential implementation, aside from very small numerical inaccuracies.

The execution time of the kernel implementation is shown in Table J.1. It is found that for a small number of training examples it is slower than the CPU implementation but it becomes faster with more training examples. This is shown in Figure 9.5.

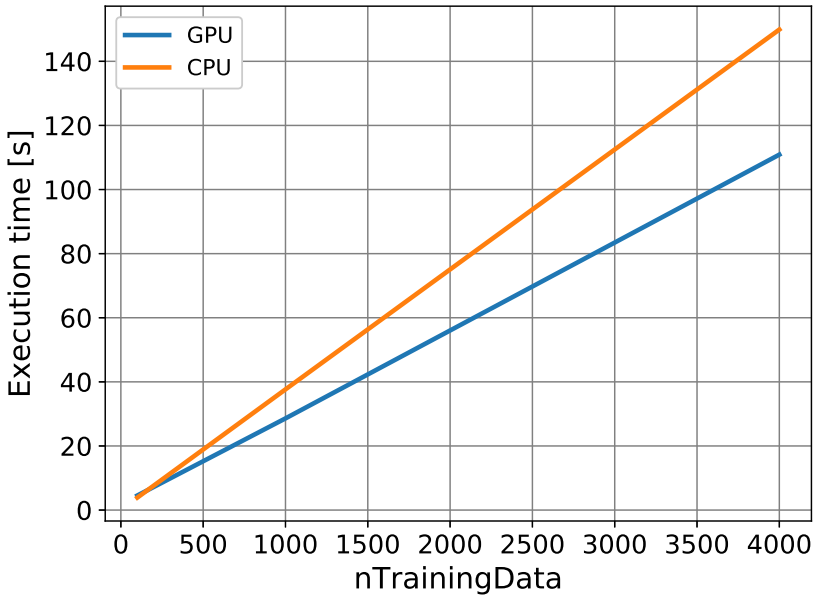


Figure 9.5: Execution time of the kernel implementation compared with that of the sequentially optimized implementation.

The code was line profiled in subsection J.5.2 and showed that the computation of both the matrix and vector projection took ∼40 % of the total time when using a larger number of training examples. The NVIDIA CUDA profiling showed that 55 % of that time was data transfer to and from the GPU. This implementation also transfers all arrays to the

GPU and back instead of creating local arrays for parts of the computation like the LU decomposition. Furthermore, it transfers all the training examples back and forth for each iteration of t even though these do not change between iterations. This could be limited to potentially show a larger decrease in execution time.

9.6 Second Iteration of Custom CUDA Kernel Implementation

The line profiling of the kernel implementation in subsection J.5.2 showed that most of the time was spent on updating the dictionary after the matrix and vector projections were computed. The second most time consuming part was data transfer between the CPU and GPU as evident by the line profiling and the NVIDIA profiling in subsection J.5.4. There is thus a large potential time decrease by updating the dictionary on the GPU in parallel and limit the amount of data to transfer. By doing this, the projection matrices does not need to be transferred back as they are only required for updating the dictionary. This section will implement another CUDA kernel to update the dictionary i.e. the sum for all n in Equation (4.1b). It will also optimize the other kernel in terms of data transfer and accessing data.

9.6.1 Reducing Data Transfer for Matrix and Vector Projection Kernel

The variables that are transferred to the GPU and back again are shown as the parameters of the kernel function in Listing 9.10.

Listing 9.10: Code snippet that shows the variables of the matrix and vector projection kernel.

```
1 @cuda.jit
2 def k_matvecproj(M, N, K, S, D, DtD, I_D, DtY, Y, vecproj, L, U, z, x, matproj):
```

As M, N, K, and S are static global variables, they can be omitted because the JIT compiler will replace them with their values when the function is compiled. Currently all the variables are transferred to the GPU when the function is invoked and back again when the function completes. All the training examples in Y should only be transferred to the GPU once and not back again. This is done by using the cuda.to_device() function from Numba. Similarly when the other kernel is implemented there is no need to transfer the empty arrays DtY, vecproj, and matproj to and from the GPU. They are instead allocated on the GPU once and overwritten in each iteration of t when they are computed. The allocation is done using the cuda.device_array(shape) function from Numba.

The variables $\boldsymbol{D}$, $\boldsymbol{D}^T\boldsymbol{D}$, and $\boldsymbol{I_D}$ will be allocated once and they will be transferred to the GPU once for each iteration of t but not back again. Finally L and U will not be transferred at all but instead each thread will allocate local memory using the cuda.local.array(shape, dtype) function from Numba in the kernel function. A quick test

showed that doing the same for z and x inexplicably increased the total execution time by over 20 % with $N = 4000$, so instead they are allocated once by the CPU and overwritten when computed in each iteration of t. This time increase manifested almost entirely in the first iteration of t. This suggests that it has something to do with the JIT compilation of the kernel function. Why the addition of the two extra local variables increases the compilation time drastically is out of scope for this project and the workaround is used instead.

Finally, as the computation of the signer variable has to be used for the kernel that computes the updated dictionary, it is also allocated on the GPU and is computed in the first kernel. All these changes are shown in Listing 9.11.

Listing 9.11: Code snippet of memory optimization for matrix and vector projection kernel.

```
### Initialization of ITKrM algorithm prior to this
d_Y = cuda.to_device(Y.reshape(-1)*1)
d_vecproj = cuda.device_array(shape=(N*M*S))
d_DtY = cuda.device_array(shape=(N*S))
z = cuda.device_array(shape=(N*S))
x = cuda.device_array(shape=(N*S))
d_matproj = cuda.device_array(shape=(M*N))
d_signer = cuda.device_array(shape=(S*N))

for t in range(maxit):
    for n in range(N):
        I_D[:,n] = np.argpartition(np.abs(D.T@Y[:,n]), -S)[-S:]
    D_new = np.zeros((M, K))
    DtD = D.T@D

    d_D = cuda.to_device(D.reshape(-1)*1)
    d_DtD = cuda.to_device(DtD.reshape(-1)*1)
    d_I_D = cuda.to_device(I_D.reshape(-1)*1)
    d_Dnew = cuda.to_device(D_new.reshape(-1)*1)

    k_matvecproj[TpB, BpG_N](d_D, d_DtD, d_I_D, d_DtY, d_Y, d_vecproj, z, x,
                             d_matproj, d_signer)

    ### Rest of ITKrM algorithm is here

@cuda.jit
def k_matvecproj(D, DtD, I_D, DtY, Y, vecproj, z, x, matproj, signer):
    n = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    # Local arrays
    L = cuda.local.array(shape=SS, dtype=nb.float64)    # SS = S*S
    U = cuda.local.array(shape=SS, dtype=nb.float64)
    ### Rest of kernel function below this
```

Furthermore there is a lot of optimization to be done w.r.t. accessing global memory, and utilizing shared memory and registers for faster data access [37]. This is a rather large subject and will not be investigated heavily, however whenever there are sums in for-loops it will be changed to save it in a temporary variable. The temporary variables are saved in registers [38] and used instead of overwriting in global or local memory which

is assumed to be faster. Finally, it is seen in Listing 9.7 and Listing 9.8 that computing vecproj, the LU decomposition, and solving it afterwards, each have an outer-most loop over S. These extra loops can simply be removed which results in a different order of the computations but the same result. It is expected that fewer loop initializations is faster. These changes are shown in Listing 9.12.

Listing 9.12: Code snippet of matrix and vector projection kernel using temporary variables.

```
@cuda.jit
def k_matvecproj(D, DtD, I_D, Y, vecproj, z, x, matproj, signer):
    n = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x

    # Local arrays
    L = cuda.local.array(shape=SS, dtype=nb.float64)
    U = cuda.local.array(shape=SS, dtype=nb.float64)
    if (n < N):
        ### Vector projection
        for i in range(0, S):
           # Calculate D^T[s,:] @ Y[:,n]
            DtY = 0
            for m in range(0, M):
                DtY += D[I_D[n + i*N] + m*K] * Y[n + m*N]

            # Calculate D/diag(DtD)*DtY
            for m in range(0, M):
                vecproj[i + m*S + n*M*S] = D[I_D[n + i*N] + m*K] * DtY / DtD[I_D[n + i*N] +
                                           I_D[n + i*N]*K]

        ### Signer
            signer[n + i*N] = math.copysign(1, DtY)

        ### LU decomposition
        # for i in range(0, S):
            for j in range(i, S):
                tmp = DtD[I_D[n + i*N] + I_D[n + j*N]*K]
                for k in range(0, i):
                    tmp -= L[k + j*S] * U[i + k*S]
                L[i + j*S] = tmp

            U[i + i*S] = 1
            for j in range(i+1, S):
                tmp = DtD[I_D[n + j*N] + I_D[n + i*N]*K] / L[i + i*S]
                for k in range(0, i):
                    tmp -= (L[k + i*S] * U[j + k*S]) / L[i + i*S]
                U[j + i*S] = tmp

        ### Solve system
        # for i in range(0, S):
            tmp = DtY / L[i + i*S]
            for j in range(0, i):
                tmp -= (L[j + i*S] * z[j + n*S]) / L[i + i*S]
            z[i + n*S] = tmp

        for i in range(S-1, 0-1, -1):
            tmp = z[i + n*S]
            for j in range(i+1, S):
```

```
48                tmp -= U[j + i*S] * x[j + n*S]
49            x[i + n*S] = tmp

51        ### D@x
52        for m in range(0, M):
53           tmp = 0
54           for s in range(0, S):
55               tmp += D[I_D[n + s*N] + m*K] * x[s + n*S]
56            matproj[m + n*M] = tmp
```

This concludes a small memory optimization for the matrix and vector projection kernel. Now a kernel will be implemented, that updates the dictionary using the projections.

9.6.2 Implementing a Dictionary Updating Kernel

In order to implement the dictionary updating kernel it is important to first find out which parts can be executed in parallel. To find out, Equation (4.1b) is rewritten as an entry-wise computation in Equation (9.7).

$$\begin{aligned} \text{For} \quad & n = 0, ..., N-1 \qquad m = 0, ..., M-1 \qquad s = 0, ..., S-1 \\ & \bar{d}_{m,i_s} \mathrel{+}= (y_{m,n} - q_{m,n} + v_{m,i_s,n}) \operatorname{sign}(d_{m,i_s} \cdot y_{m,i_s}) \end{aligned} \tag{9.7}$$

Where i_s is also dependent of n as it is the s-th entry in $\boldsymbol{I}^t_{\boldsymbol{D},n}$. Given this equation it is possible to see the following possibilities for parallel computation of updating the dictionary:

- Each training example n.
- Each row m of $\boldsymbol{D}$.
- Each element s in the index set.
- Each column k of $\boldsymbol{D}$.

It is not immediately obvious which part to implement in parallel is ideal. Generally it would be best to have the loop with the highest number of iterations implemented in parallel in order to fully utilize all the processors available, in an attempt to avoid the case where the number of iterations is smaller than the available processors. For this problem it is a requirement that $N > K > S$ while M is independent of their values.

There is however a major issue with making N parallel threads which is that each processor needs to write to the i_s-th column of $\bar{\boldsymbol{D}}$. As $N > K$ it is guaranteed that some columns are represented multiple times in the index set $\boldsymbol{I}^t_{\boldsymbol{D},n}$. And while it is not a guarantee that each processor is writing to the same row of $\boldsymbol{D}$ at the same time, there is a larger than 0 % chance of two processors trying to write to the same element at the same time. This

is a critical flaw as it could lead to an incorrect computation. Implementing S parallel threads has the exact same flaw.

It is possible instead to have each processor updating each column k of the dictionary in parallel. This avoids the issue of writing to the same address of memory simultaneously. It does however present a new problem in that it is necessary to know which n and s results in an element equal to k in the index set $\boldsymbol{I}^t_{\boldsymbol{D},n}$. So in order to update each column in parallel there is some extra work to find the n where $k \in \boldsymbol{I}^t_{\boldsymbol{D},n}$.

The last option is to compute each row m of $\boldsymbol{D}$ in parallel. Again there is no issue of writing to the same address simultaneously as each processor will handle its own row. It also avoids the issue with updating each column as it can simply compute for all n sequentially and find the correct column in the index set directly. The only issue with this implementation is that the hardware utilization is entirely dependent on the image size and some problems are not necessarily scalable with more processors.

The last option will be the way that the kernel, that updates the dictionary, will be implemented as it avoids all the major flaws and issues at the cost of poor or difficult scalability. The kernel function is computed based on Equation (9.7) and is shown in Listing 9.13.

Listing 9.13: Implementation of a dictionary updating kernel.

```
@cuda.jit
def k_updateD(D, I_D, Y, vecproj, matproj, signer):
    m = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    if m < M:
        for n in range(0, N):
            for s in range(0, S):
                D[I_D[n + s*N] + m*K] += (Y[n + m*N] - matproj[m + n*M] +
                                         vecproj[s + m*S + n*M*S])*signer[n + s*N]
```

Lastly it needs to return the updated dictionary to the CPU which is done by using the .copy_to_host() command on an array located on the GPU. The complete ITKrM algorithm using the CUDA kernels is shown in Listing 9.14.

Listing 9.14: ITKrM algorithm implementation using CUDA kernels.

```
def gpu_itkrm(data, K, S, maxit):
    M, N = data.shape
    D_init = np.random.randn(M, K)
    for i in range(K):
        D_init[:,i] = D_init[:,i] / np.linalg.norm(D_init[:,i], 2)
    Y = data
    I_D = np.zeros((S, N), dtype=np.int32)
    D = D_init
    TpB = 32    # ThreadsPerBlock
    BpG_N = math.ceil(N/32)     # BlocksPerGrid
    BpG_M = math.ceil(M/32)

    # Move training data to device and allocate arrays on device.
```

```
14      d_Y = cuda.to_device(Y.reshape(-1)*1)
15      d_vecproj = cuda.device_array(shape=(N*M*S))
16      z = cuda.device_array(shape=(N*S))
17      x = cuda.device_array(shape=(N*S))
18      d_matproj = cuda.device_array(shape=(M*N))
19      d_signer = cuda.device_array(shape=(S*N))

21      for t in range(maxit):
22          for n in range(N):  # Index set
23              I_D[:,n] = np.argpartition(np.abs(D.T@Y[:,n]), -S)[-S:]
24          D_new = np.zeros((M, K))
25          DtD = D.T@D

27          d_D = cuda.to_device(D.reshape(-1)*1)
28          d_DtD = cuda.to_device(DtD.reshape(-1)*1)
29          d_I_D = cuda.to_device(I_D.reshape(-1)*1)
30          d_Dnew = cuda.to_device(D_new.reshape(-1)*1)

32          #   Calculates matrix and vector projection
33          k_matvecproj[TpB, BpG_N](d_D, d_DtD, d_I_D, d_Y, d_vecproj, z, x, d_matproj, d_signer)

35          #   Update dictionary
36          k_updateD[TpB, BpG_M](d_Dnew, d_I_D, d_Y, d_vecproj, d_matproj, d_signer)
37          D_new = d_Dnew.copy_to_host()
38          D_new = D_new.reshape(M, K)

40          scale = np.sum(D_new*D_new, axis=0)
41          iszero = np.where(scale < 0.00001)[0]
42          D_new[:,iszero] = np.random.randn(M, len(iszero))

44          #   Normalize dictionary
45          D_new = normalize_mat_col(D_new)
46          D = 1*D_new
47      return D
```

9.6.3 Evaluation of the Second Iteration of CUDA Kernel Implementation

The 2nd iteration using CUDA kernels is tested and the tests are shown in Appendix K. The implementation is tested for correctness in subsection K.5.1 by checking that it produces an identical dictionary as the previous implementation aside from very small numerical inaccuracies.

The execution time of this implementation is shown in Table K.1. It is found that the implementation is faster than the CPU for all the different amount of training data that was tested. For larger amount of training data this implementation is more than 7 times faster than the sequential implementation. The execution times of the custom CUDA kernel implementations are shown in Figure 9.6 in relation to the sequential implementation.

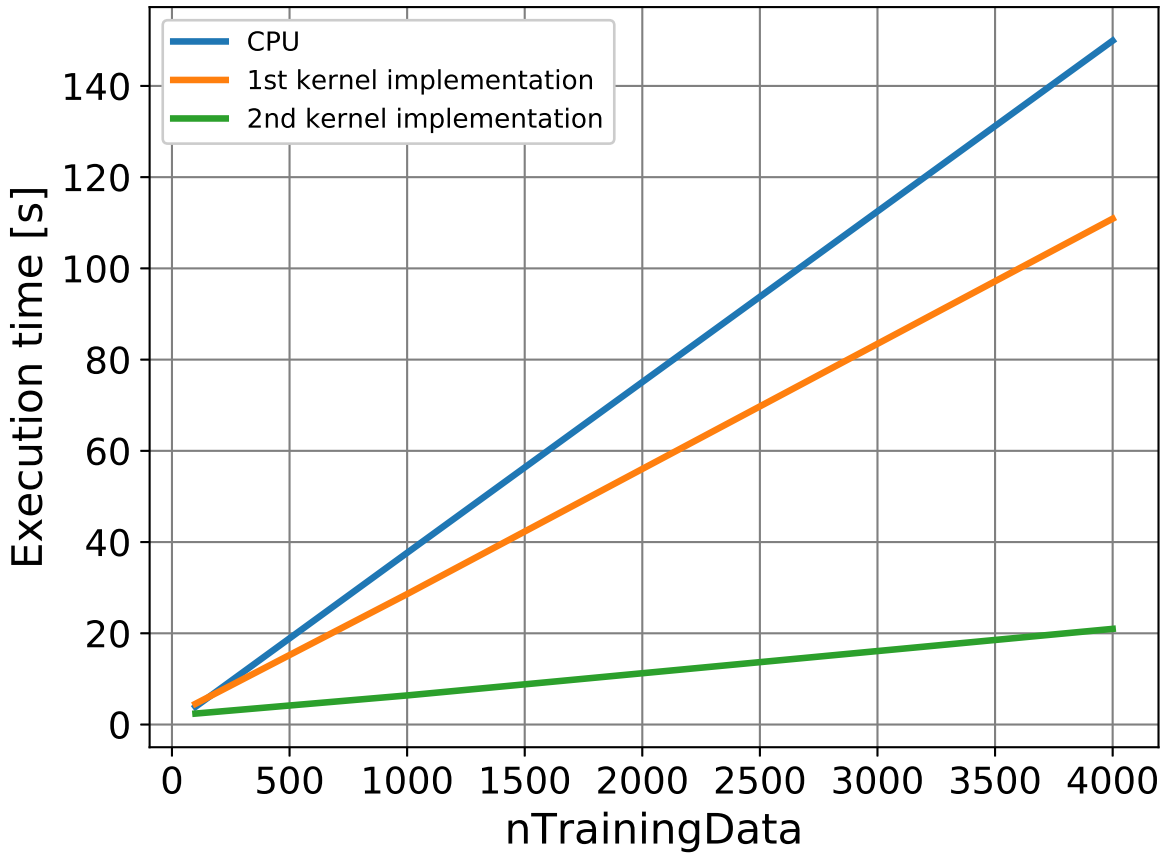


Figure 9.6: Execution time of the kernel implementation compared with that of the sequentially optimized implementation.

The code was line profiled in subsection K.5.2 and showed that roughly 60 % of the time is spent on the GPU part of the code and most of the remaining time is spent on computing the index set. The NVIDIA profiling in subsection K.5.4 showed that less than 5 % of the GPU part was data transfer.

9.7 Discussion of the Parallel Optimization using GPU

It was shown that there were a couple of different options when it comes to implementing the ITKrM algorithm in parallel on a GPU. This could be each entry of a matrix operation computed in parallel or computing a sub-dictionary for each training example in parallel.

CuPy was used to implement the first option, and the custom CUDA kernel was an implementation of the second option, but required a lower abstraction level in comparison. The CuPy implementation did not show an improvement in execution time, despite of the many more processing cores available on the GPU. It is unsure exactly why that is the case but using a high level module with built-in functions could give some overhead.

There is also a large number of function calls in this implementation as the dictionary updating step is computed in a for-loop. This could also be part of the reason that the CuPy implementation does not show a decrease in execution time compared to an implementation on the CPU.

The second CUDA kernel implementation using Numba did show an execution time that was around 7 times faster with a given problem size, a specific CPU, and a specific GPU. This implementation utilized the inherent parallelism in the algorithm itself rather than for each linear algebra operation. It was however not as intuitive as the CuPy implementation as it required to write all the operations. This included matrix-vector products, matrix-matrix products, handling matrix indexing, and solving a system of linear equations using LU decomposition. All of these operations had to be explicitly coded which leads to an increase in development time in comparison to using CuPy. Furthermore the CUDA kernel implementation also required deeper insight into the GPU architecture and memory handling to get a faster execution time.

It is not worth implementing the ITKrM algorithm on the GPU using the CuPy module as it lead to a slower execution time. The custom CUDA kernel implementation lead to a faster execution time, but whether the extra development time was worth it, depends entirely on the problem size.

Part III

Test and Comparison

10 | Comparison between Implementation Strategies

Through Chapter 7, Chapter 8 and Chapter 9 three different approaches to optimizing the ITKrM algorithm have been made. This chapter compares the results from the sequential optimization, the parallel optimization on CPU and the parallel optimization on GPU. As stated in section 6.1 the initial idea was to test all implementations on the Phoenix 1 node in the Birdnest cluster, with the purpose of being able to make an exact comparison between the different implementations. Due to the problems mentioned in section 8.4 and section 9.1, only the sequential implementations were tested on the Phoenix node. To be able to still make a valid comparison the tests of the parallel CPU and the parallel GPU implementations were compared to the 3rd iteration of the sequential optimization on the same unit under testing. It is assumed that the ratio between execution times of different implementations will remain the same on different CPUs. For example if a comparison is made between the sequential optimized implementation and the parallel CPU optimized implementation on a unit A, then the relation between the two implementations execution times will be the same on a unit B. The sequential optimization, from naïve to the 3rd iteration, showed a change in time from 2623.16 s to 6.38 s. This is a huge improvement, most likely because the naïve implementation was mapped directly from the mathematical description, which is not optimized computationally, but easier to understand. The time reduction achieved makes the sequential optimization part quite successful, and the focus will now be on the rest of the optimization. Figure 10.1 shows a comparison of execution time between the 3rd sequential optimized- and the parallel CPU optimized- implementations. The figure is from a test made in Appendix E.

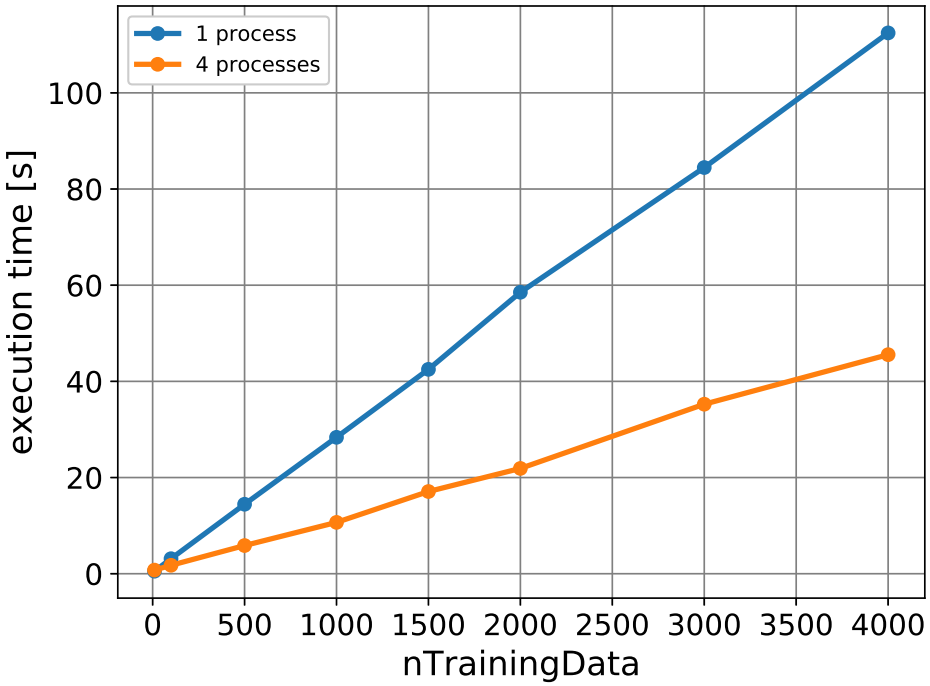


Figure 10.1: Execution time using 1 and 4 processes.

Figure 10.1 shows that for all values of nTrainingData the use of 4 processes on CPU is faster than the 3rd sequential implementation. For nTrainingData = 1000 the parallel CPU implementation with 4 processes executes 2.66 times as fast as the 3rd sequential implementation, and for nTrainingData = 4000 the parallel CPU implementation with 4 processes executes 2.47 times faster than the 3rd sequential implementation. Thus it can be evaluated that the parallel CPU implementation has reduced the execution time of the ITKrM algorithm compared to the 3rd sequential implementation.

Three different implementations of the ITKrM algorithm have been made with the use of a GPU. From Figure 9.4 in Chapter 9 it is seen that the naïve implementation of the ITKrM algorithm with the use of a GPU had a higher execution time than the 3rd sequential implementation. To optimize the naïve GPU implementation a custom CUDA kernel was developed. Figure 10.2 shows a comparison of execution time between the 3rd sequential implementation and the 1st and 2nd implementation of the GPU custom CUDA kernel.

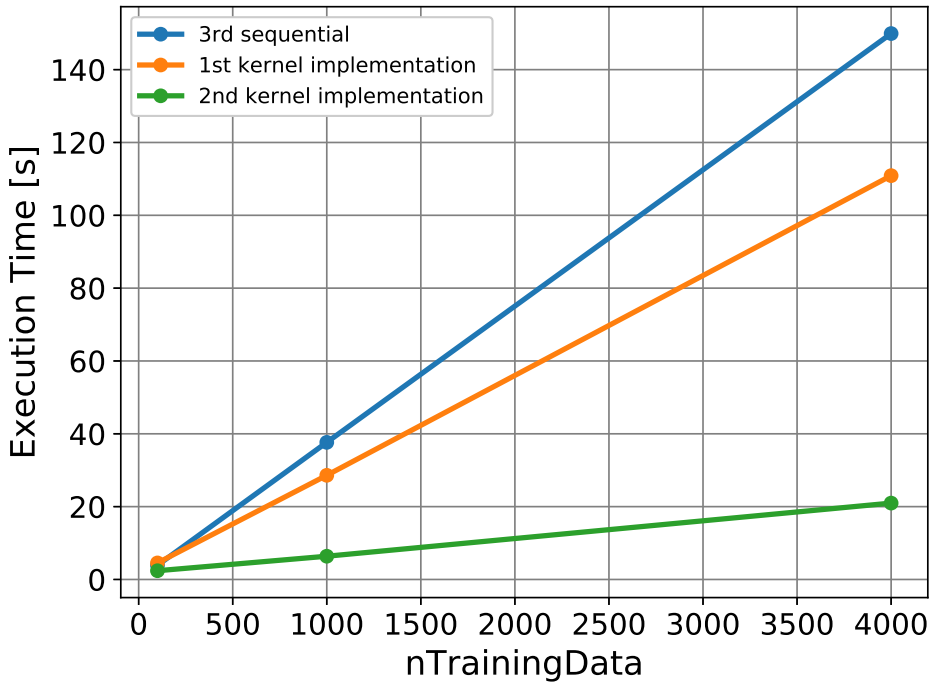


Figure 10.2: Execution time of the CUDA kernel implementation compared with that of the sequentially optimized implementation.

Figure 10.2 shows that for all values of nTrainingData the 2nd kernel implementation is the fastest. For nTrainingData = 1000 the 2nd GPU implementation executes 5.90 times faster than the 3rd sequential implementation, and for nTrainingData = 4000 the 2nd GPU implementation executes 7.15 times faster than the 3rd sequential implementation. When comparing the results in Figure 9.4 and Figure 10.2 it can be seen that the use of a GPU can be a great improvement if handled correctly. If not handled correctly the use of a GPU might lead to a reduction in performance. Thus it can be evaluated that the 2nd kernel implementation has reduced the execution time of the ITKrM algorithm compared to the 3rd sequential implementation.

To compare the 3rd sequential, the parallel CPU, and the 1st and 2nd kernel implementation the ratio between the execution times are used. This is done because the results from Figure 10.1 were made in a test on one device, and the results from Figure 10.2 were made on another device. The execution times of the parallel CPU implementation with 4 processes in Figure 10.1 is put in relation to the execution times of the 3rd sequential implementation in Figure 10.1. These relations are then used to visualize the execution times of the parallel CPU implementation with 4 processes in relation to the 3rd sequential and the 1st and 2nd kernel implementation in Figure 10.2. Figure 10.3 shows the comparison of execution time between the 3rd sequential, the parallel CPU, and the 1st and 2nd GPU implementation.

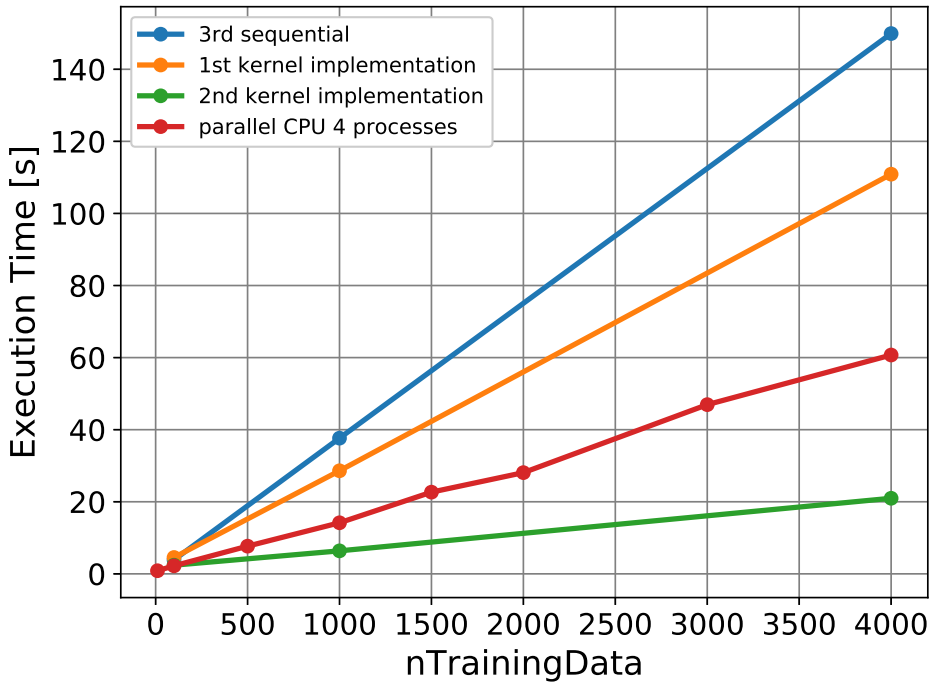


Figure 10.3: Execution time comparison between the 3rd sequential, the parallel CPU, and the 1st and 2nd kernel implementations.

Figure 10.3 shows that the 2nd kernel implementation is the fastest of all the implementation made through this project. At nTrainingData = 4000 the 2nd kernel implementation is three times as fast as the parallel CPU implementation with 4 processes. The 2nd kernel implementation is able to form a dictionary from 4000 training signals in 20.97 s.

Figure 10.3 also shows that the parallel CPU implementation with 4 processes is 1.83 times as fast as the 1st kernel implementation. This is interesting to notice as this again supports that the use of a GPU must be handled with care as it does not guarantee an improvement in execution time. Through this project, two optimization iterations had to be made to make the GPU perform better than the CPU implementations, but once done correctly the use of a GPU made the largest reduction in execution time of the ITKrM algorithm.

11 | Visual Performance

The performance evaluation of the ITKrM algorithm has up till now been with a focus on the computational performance and the execution time. This chapter evaluates on the visual performance of the ITKrM algorithm, specifically on the relation between the SSIM index and the size of the sub-images used to train the algorithm. As said in subsection 4.3.1 the SSIM index is a measure of the perceived quality of a digital image. section 4.3 discusses the influence of the size of the sub-images. Large sub-images gives large atoms and can make it difficult to capture small attributes of an image. Smaller sub-images gives smaller atoms and is more fitting for capturing small attributes. The smaller sub-images comes with the cost of less compression if the sparsity level S is held constant. This chapter aims to get an idea of how the size of the sub-images affects the visual performance. In subsection 4.3.1 the images used for both training the ITKrM algorithm and testing have the shape 32×32, which is considered a small image. This fact limits the possibility of dividing the images into smaller sub-images and therefore some larger images are used for evaluating the visual performance. The used images are MRI scans [39] with the dimensions 128×128.

In Appendix L a test of the visual performance is made, where the test is divided into two sub-tests. One where the sparsity level S follows the size of the sub-images and one where S is held constant. In both tests S are the same in the process of finding a dictionary in the ITKrM algorithm and when finding the sparse solution in the OMP algorithm, and the number of atoms K in the dictionary $\boldsymbol{D}$ is held constant at 200. The SSIM index is found by comparing an original image with an image restored from the sparse solution of the original image, which is the case in both tests. The test with the relative sparsity level is made to find the isolated effect of the size of the sub-images. The sparsity level is divided by four every time the dimensions of the sub-images is divided by two. This is because as a sub-image goes from dimensions 64×64 to 32×32 the training set will contain four times as many images, and thus the sparsity level should, for example, go from 80 to 20. This way the number of non-zero elements in the found sparse solutions will remain the same relative to the subpicture size. In Figure 11.1 a relation is shown between the dimension of the sub-images and the SSIM index with a relative sparsity level.

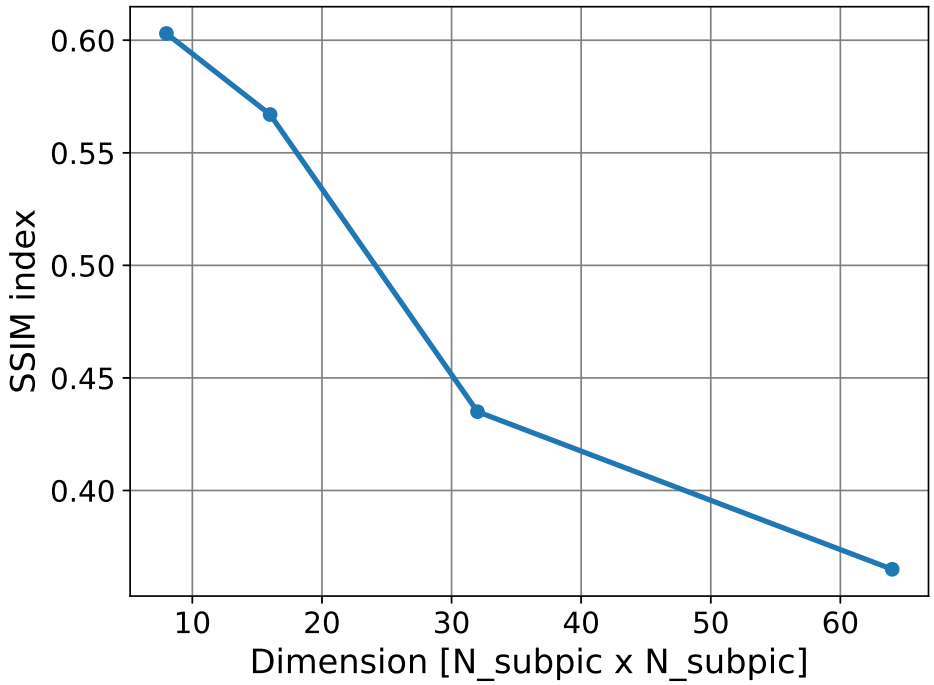
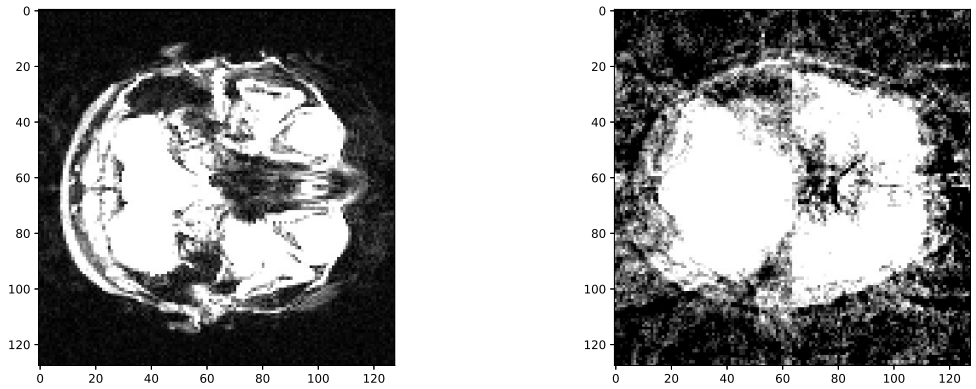


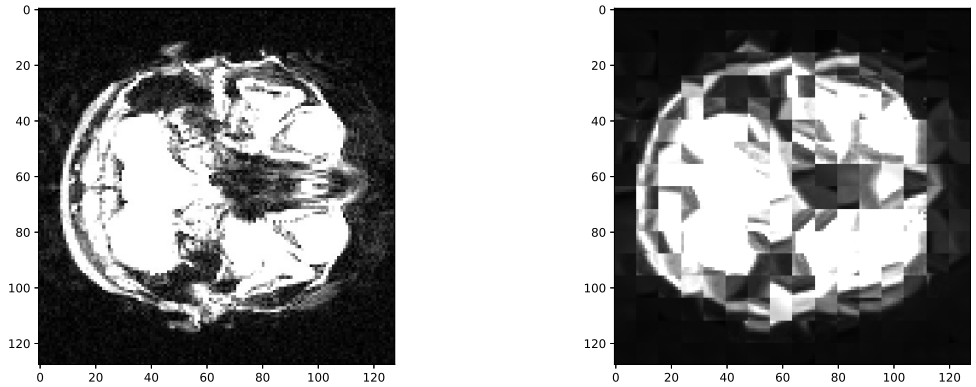
Figure 11.1: Relation between dimension [N_subpic × N_subpic] of sub-images and SSIM index with relative sparsity level.

Figure 11.1 shows that with N_subpic×N_subpic dimensions of the sub-images, the SSIM index increases as N_subpic decreases. The SSIM index lies within 0.37 and 0.60. These values are relatively low, but it should be noticed that for example when N_subpic = 16 the sub-images will consist of 256 entries, and only five values are used in the sparse solution, meaning a reduction of 51.2 times. Figure 11.2 and Figure 11.3 shows the results of the visual performance test with N_subpic = 64 and 8 with relative sparsity level.



(a) beforeImage **(b)** afterImage with N_subpic = 64 and S = 80

Figure 11.2: Results visual performance test with N_subpic = 64 and S = 80. SSIM index = 0.365



(a) beforeImage **(b)** afterImage with N_subpic = 8 and S = 1

Figure 11.3: Results visual performance test with N_subpic = 8 and S = 1. SSIM index = 0.603

From Figure 11.2b it is seen that the afterImage is formed from many small elements, and that the restored image of the beforeImage is not very precise, which is also seen in the SSIM index. The afterImage in Figure 11.3b is formed from relatively few large elements. The restored image of the beforeImage is still not very precise, but the SSIM index indicates that the restoring is better for smaller dimensions of the sub-figures.

The test with the constant sparsity level is made to evaluate how it affects the SSIM index when the sparsity level goes towards the total number of elements in the sparse solutions. In Figure 11.4 a relation is shown between the dimension of the sub-images and the SSIM index with a constant sparsity level of $S = 40$.

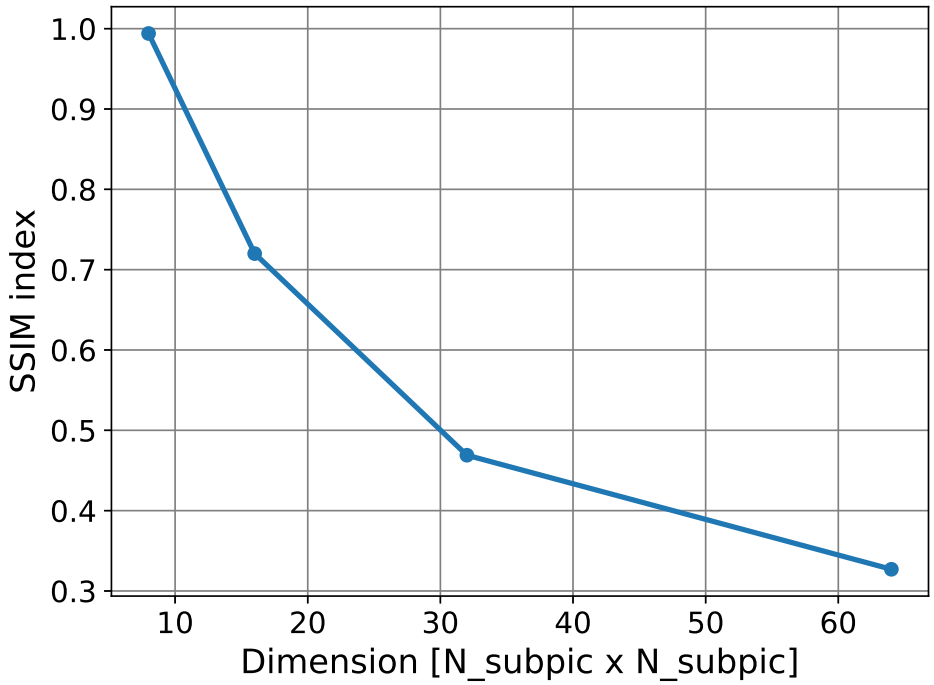
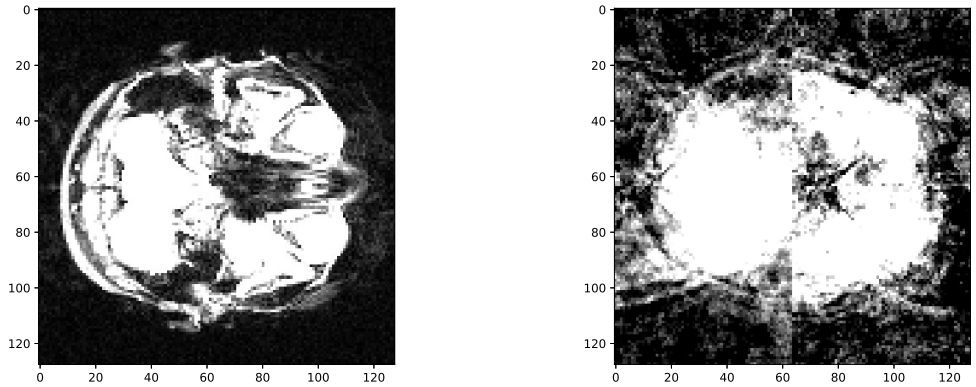


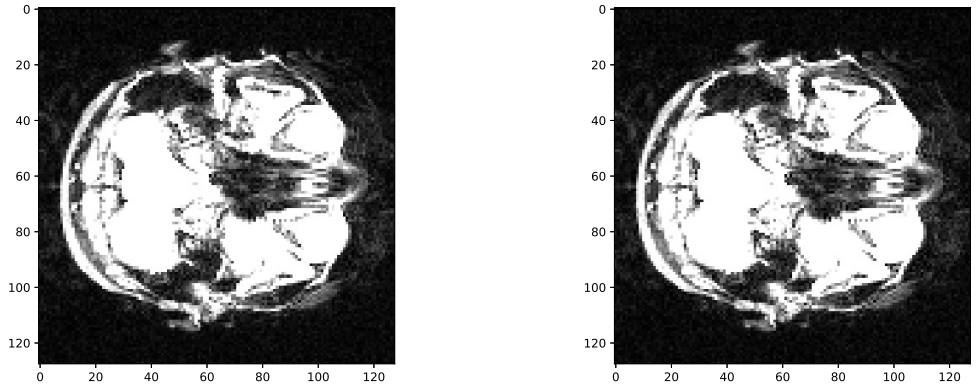
Figure 11.4: Relation between dimension [N_subpic x N_subpic] of sub-images and SSIM index with a constant sparsity level $S = 40$.

From Figure 11.4 it is seen that with N_subpic x N_subpic dimensions of the sub-images, the SSIM index increases as N_subpic decreases. The SSIM index lies within 0.33 when N_subpic = 64, and 0.99 when N_subpic = 8, in both cases with a sparsity level of 40. In the case with N_subpic = 64 and S = 40 the sparse solutions will contain 40 non-zero elements out of a total of $64 \cdot 64 = 4096$ elements, meaning that $\frac{40}{4096} \cdot 100 = 0.97\%$ of the data is non-zero. In the case with N_subpic = 8 and S = 40 the sparse solutions will contain 40 non-zero elements out of a total of $8 \cdot 8 = 64$ elements meaning that $\frac{40}{64} \cdot 100 = 62.5\%$ of the data is non-zero. This fact explains why the SSIM index reaches much higher values for a constant sparsity level than for a varying sparsity level. These percentages can be seen as an index of how much the original images are compressed, but without taking into account if the compression is lossless. Figure 11.5 and Figure 11.6 shows the results of the visual performance test with N_subpic = 64 and 8 with a constant sparsity level $S = 40$.



(a) beforeImage **(b)** afterImage with N_subpic = 64 and $S = 40$

Figure 11.5: Results visual performance test with N_subpic = 64 and $S = 40$. SSIM index = 0.327



(a) beforeImage **(b)** afterImage with N_subpic = 8 and $S = 40$

Figure 11.6: Results visual performance test with N_subpic = 8 and $S = 40$. SSIM index = 0.994

The afterImage in Figure 11.5b shows that the restored image of the beforeImage when N_subpic = 64 is not very precise, but it is seen in Figure 11.6b that the restored image of the beforeImage is almost identical to the beforeImage, which is also seen in the SSIM index. This supports the idea that with more non-zero entries in the sparse solution, the better a restored image can be obtained. The full visual performance test can be seen in Appendix L.

The visual performance test showed that the size of the sub-images and the amount of non-zero data in the sparse solution have a large influence on the visual perception of the restored images. This is a fact that should be taken into consideration when using the ITKrM to train dictionaries as both the sparsity level and the number and size of the training data are a part of the computations in the algorithm.

11.1 Discussion of the Visual Performance

In the test of the visual performance, only two parameters were investigated in relation to their effect on the visual performance. The two were the sparsity level S and the size of the [N_subpic x N_subpic] shaped sub-images. It was seen that a higher sparsity level relative to the size of the sub-images improved the visual performance. This improvement comes at the cost of less compression and thus these parameters should be balanced in order to meet specific requirements. In Appendix L two tests were performed, but many other tests regarding visual performance could be made as the setting of parameters in the ITKrM algorithm is a very complex task. Parameters that could also be tested regarding their influence on the visual performance could be the number of atoms K in the dictionary $\boldsymbol{D}$ or the number of iterations t in the ITKrM algorithm. These parameters are not investigated as the main focus of the project is the optimization of the ITKrM, and not the actual quality of the results produced by the ITKrM algorithm. It was although of interest to give a brief view of how some of the parameters in the ITKrM algorithm affect the visual performance.

Part IV

Discussion and Conclusion

12 | Discussion

There are some considerations that should be made when implementing an algorithm. How often is this algorithm going to be used? Does it only have to run one time in total or is it run 10,000 times a day? A balance should exist between the time spent on optimization and how often the algorithm is used. The execution time of a dictionary learning algorithm has through this project been optimized in a sequential and parallel fashion on a CPU, and in a parallel fashion on a GPU in combination with a CPU. The focus of the project has been the optimization of the algorithm and the comparison between the different implementations of the algorithm.

The optimization from the naïve sequential CPU implementation to the last custom kernel GPU-CPU combined implementation resulted in an execution time which was 2928 times faster. However, it can be argued that the time used on optimizing the implementation in this specific case was too high compared to the frequency in which the algorithm will be used. A dictionary learning algorithm is only used when a dictionary is trained, hereafter an algorithm such as the OMP uses the dictionary to find a sparse representation. The dictionary learning algorithm is therefore rarely used, as it is only used if more or completely new training data is introduced. However, keep in mind that the amount of data in this project is fairly small. If a dictionary learning algorithm was used for a larger dataset, then the time used on optimizing can be argued to be sensible, as execution time would increase considerably with larger datasets. The necessity of optimization varies from case to case, where the frequency of use should be considered together with the amount of training data used.

An initial decision was to use the Phoenix compute node in the Birdnest compute cluster for both the CPU and GPU implementations. In addition to this, it was decided to write as much code in Python as possible. However, it became evident that these two decisions could not be achieved in combination. It was investigated which modules could be used to implement a simple version of the ITKrM algorithm on the GPU. Here it was found that the most favourable module was the CuPy module, as CuPy for instance easily enabled the use of inverse and pseudo-inverse. However, after an initial setup, an incompatibility was found between the compute capability of the Phoenix node and CuPy. It should be noted that the later iterations of the GPU implementation uses Numba, which can be run on the Phoenix compute node. The problems found when running the parallel CPU implementation on the Phoenix compute node that was found in Appendix F, meant that a valid comparison of the parallel CPU vs. the parallel GPU implementation could not be made on the Phoenix compute node, as the results from running the parallel CPU implementation did not behave as expected. Therefore the GPU tests were not performed on the Birdnest cluster. An additional drawback of using the Birdnest cluster is the difficulties of developing code on the cluster. These development difficulties are for example a lack of interface for variable exploring and a lack of plotting tools, which makes the process of debugging more difficult. This was not the reason why the Birdnest cluster

was not used as intended, but it is a subject for consideration.

A subject for discussion when using GPUs is the different specifications for different GPUs. These differences includes the single vs. double precision performance on different GPUs. A specific GPU may be developed for single precision operations and not double precision operations or vice versa. In relation to this it could be discussed if it would be reasonable to implement the different GPU implementations of the ITKrM algorithm in single precision instead of double precision. This might lead to a faster execution time although it might also introduce larger differences in the computed dictionaries of the different GPU implementations.

It should be considered that the optimization process is a subjective process, where each optimization step evolves from a programmers idea. Other directions could have been taken when specific implementations were optimized. More time could be used on optimizing the different lines of code and optimizing the individual implementations. With more time it would have been interesting to explore the implementation of the custom CUDA kernel more in-depth w.r.t. utilizing shared memory and memory access in general. The potential for performance improvement is quite large as evident in [37]. There was also a possibility of looking into vectorizing the sequential implementation further or using an extra dimension in an attempt to avoid the loop for all training examples. It could also be discussed if the decision about using Python as the main implementation language was beneficial. It might have revealed more parameters to optimize if it was chosen to use a lower level programming language than Python. This would especially be relevant in cases where the execution time is the optimization parameter.

No specific requirements were made to the execution time of the ITKrM algorithm, thus making it difficult to evaluate if the algorithm was sufficiently optimized. This is accepted as the aim of the project was to investigate how different implementation methods and hardware platforms would alter the execution time of the ITKrM algorithm and its difficulty of implementation.

13 | Conclusion

Part I, Pre-analysis, investigated some different fields of study within scientific computing which was done according to the initial problem statement in section 1.1. Part I provided insights which were used to formulate the problem statement:

How do different implementation methods and hardware platforms alter the execution time of the ITKrM algorithm and its difficulty of implementation?

This was answered by first describing and testing the different implementations. The three sequential optimizations on a CPU reduced the execution time of the ITKrM algorithm by a factor 410. The sequential CPU optimization was followed by a parallel optimization on a CPU. Here it was possible to reduce the execution time of the ITKrM algorithm an additional 3.7 times with the use of four processes instead of one. The last implementation method combined a CPU and a GPU, where the final CUDA kernel implementation reduced the execution time by a factor 7.1 compared to the sequential implementation. The sequential optimization does however require less hardware available. It should be mentioned that the naïve CPU-GPU implementation was slower than the fastest sequential CPU implementation and the 1st CUDA kernel implementation was slower than the parallel CPU implementation. Although the overall time reduction is substantial, it should be noted that most of the reduction came from the sequential part.

In the process of making parallel optimization it was found that a better performance w.r.t. execution time comes at a prize of an increased difficulty when implementing. These difficulties included requirements for deeper insights into specific hardware platforms and a need for a lower abstraction level than Numpy. It was found in the parallel CPU implementations that insight was needed into classifications of parallel computer architectures in order to choose the right tool for a specific problem.

It can be concluded that with the size of the data available, there is little reason to implement the ITKrM algorithm in parallel neither on the CPU nor the GPU. If the data size was significantly larger e.g. with more and/or higher resolution images, the custom CUDA kernel implementation saw the fastest execution time.

Bibliography

[1] Iliya Valchanov. Machine learning: An overview. Internet article, 2018. URL https://www.datascience.com/blog/machine-learning-overview. Accessed: 05-03-2018.

[2] Mathworks. What is machine learning. Internet article, 2018. URL https://www.mathworks.com/discovery/machine-learning.html. Accessed: 05-03-2018.

[3] Mostafa Sadeghi, Massoud Babaie-Zadeh, and Christian Jutten. Learning overcomplete dictionaries based on atom-by-atom updating. *IEEE Transactions on Signal Processing*, 62(4):883–891, Dec 2014. ISSN 1941-0476. doi: 10.1109/TSP.2013.2295062.

[4] Karin Schnass. Convergence radius and sample complexity of itkm algorithms for dictionary learning. *Applied and Computational Harmonic Analysis*, 2016. ISSN 1063-5203. doi: 10.1016/j.acha.2016.08.002.

[5] I. Tosic and P. Frossard. Dictionary learning. *IEEE Signal Processing Magazine*, 28 (2):27–38, March 2011. ISSN 1053-5888. doi: 10.1109/MSP.2010.939537.

[6] J. Yang, J. Wright, T. S. Huang, and Y. Ma. Image super-resolution via sparse representation. *IEEE Transactions on Image Processing*, 19(11):2861–2873, Nov 2010. ISSN 1057-7149. doi: 10.1109/TIP.2010.2050625.

[7] M. Nejati, S. Samavi, N. Karimi, S. M. R. Soroushmehr, and K. Najarian. Boosted dictionary learning for image compression. *IEEE Transactions on Image Processing*, 25(10):4900–4915, Oct 2016. ISSN 1057-7149. doi: 10.1109/TIP.2016.2598483.

[8] J. Wright, A. Y. Yang, A. Ganesh, S. S. Sastry, and Y. Ma. Robust face recognition via sparse representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(2):210–227, Feb 2009. ISSN 0162-8828. doi: 10.1109/TPAMI.2008.79.

[9] V. Naumova and K. Schnass. Dictionary learning from incomplete data for efficient image restoration. In *2017 25th European Signal Processing Conference (EUSIPCO)*, pages 1425–1429, Aug 2017. doi: 10.23919/EUSIPCO.2017.8081444.

[10] B. Ophir, M. Lustig, and M. Elad. Multi-scale dictionary learning using wavelets. *IEEE Journal of Selected Topics in Signal Processing*, 5(5):1014–1024, Sept 2011. ISSN 1932-4553. doi: 10.1109/JSTSP.2011.2155032.

[11] G. Rath and C. Guillemot. Sparse approximation with an orthogonal complementary matching pursuit algorithm. In *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3325–3328, April 2009. doi: 10.1109/ICASSP.2009.4960336.

[12] B. K. Natarajan. Sparse approximate solutions to linear systems. *SIAM Journal on Computing*, 24(2):227–234, 1995. doi: 10.1137/S0097539792240406. URL https://doi.org/10.1137/S0097539792240406.

[13] T. T. Cai and L. Wang. Orthogonal matching pursuit for sparse signal recovery with noise. *IEEE Transactions on Information Theory*, 57(7):4680–4688, July 2011. ISSN 0018-9448. doi: 10.1109/TIT.2011.2146090.

[14] University of Innsbruck Karin Schnass. Karin schnass - department of mathematics. Internet blog, 2018. URL https://www.uibk.ac.at/mathematik/personal/schnass/. Accessed: 12-03-2018.

[15] Alex Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, Department of Computer Science, University of Toronto, 2009.

[16] Valeriya Naumova and Karin Schnass. Dictionary learning from incomplete data. *CoRR*, abs/1701.03655, 2017. URL http://arxiv.org/abs/1701.03655.

[17] Z. Wang, E. P. Simoncelli, and A. C. Bovik. Multiscale structural similarity for image quality assessment. In *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003*, volume 2, pages 1398–1402 Vol.2, Nov 2003. doi: 10.1109/ACSSC.2003.1292216.

[18] S. Kestur, J. D. Davis, and O. Williams. Blas comparison on fpga, cpu and gpu. In *2010 IEEE Computer Society Annual Symposium on VLSI*, pages 288–293, July 2010. doi: 10.1109/ISVLSI.2010.84.

[19] Torben Larsen Thomas Arildsen Tobias Lindstrøm Jensen. *Behavioural Simulation and Computing for Signal Processing Systems*. Not yet published, 2018.

[20] Tarinder Sandhu. Review: Intel core i7 and x58 chipset. Internet article, 2008. URL http://www.hexus.net/tech/reviews/cpu/16187-intel-core-i7-x58-chipset-systems-go-fsb-invited/?page=3. Accessed: 15-05-2018.

[21] Reiner Hartenstein and Tu Kaiserslautern. *Basics of Reconfigurable Computing*, pages 451–501. Springer Netherlands, Dordrecht, 2007. ISBN 978-1-4020-5869-1. doi: 10.1007/978-1-4020-5869-1_20. URL https://doi.org/10.1007/978-1-4020-5869-1_20.

[22] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008. ISSN 0018-9219. doi: 10.1109/JPROC.2008.917757.

[23] Christoph Kubisch. Life of a triangle - nvidia's logical pipeline. Internet article, 2015. URL https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline. Accessed: 15-05-2018.

[24] Thomas Arildsen. Numerical scientific computing - computational platforms. Course material, 2018. Accessed: 15-05-2018.

[25] AMArchibald, MartinSpacek, and Pauli Virtanen. Parallel programming with numpy and scipy. Internet article, 2015. URL https://scipy-cookbook.readthedocs.io/items/ParallelProgramming.html. Acessed: 18-04-2018.

[26] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071.

[27] NVIDIA Corporation. About cuda. Internet page, 2018. URL https://developer.nvidia.com/about-cuda. Accessed: 14-05-2018.

[28] Khronos group. Opencl overview. Internet page, 2018. URL https://www.khronos.org/opencl/. Accessed: 14-05-2018.

[29] Anaconda. Numba for cuda gpus. Internet wiki, 2018. URL https://numba.pydata.org/numba-doc/dev/cuda/index.html. Accessed: 08-05-2018.

[30] Anaconda. Pyculib. Internet wiki, 2018. URL http://pyculib.readthedocs.io/en/latest/. Accessed: 08-05-2018.

[31] Pavan Yalamanchili, Umar Arshad, Zakiuddin Mohammed, Pradeep Garigipati, Peter Entschev, Brian Kloppenborg, James Malcolm, and John Melonakos. ArrayFire - A high performance software library for parallel computing with an easy-to-use API, 2015. URL https://github.com/arrayfire/arrayfire. Accessed: 08-05-2018.

[32] Prefered Networks. Cupy. Internet wiki, 2018. URL https://docs-cupy.chainer.org/en/stable/. Accessed: 08-05-2018.

[33] Mark Harris at Nvidia. Cuda pro tip: nvprof is your handy universal gpu profiler. Internet page, 2018. URL https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/. Accessed: 16-05-2018.

[34] Anaconda. Numba memory management. Internet wiki, 2018. URL https://numba.pydata.org/numba-doc/dev/cuda/memory.html. Accessed: 09-05-2018.

[35] NVIDIA. Cuda c programming guide. Webpage, 2018. URL https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. Accessed: 10-05-2018.

[36] Kartik K. Sivaramakrishnan. Lu decomposition and pivoting. Webpage, 2018. URL http://www4.ncsu.edu/~kksivara/ma505/handouts/lu-pivot.pdf. Accessed: 14-05-2018.

[37] NVIDIA Mark Harris. Optimizing parallel reduction in cuda. Webpage. URL http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf. Accessed: 17-05-2018.

[38] Ana Lucia Varbanescu. Gpu programming part 2: Advanced topics. Webpage. URL http://www.st.ewi.tudelft.nl/~varbanescu/ASCI_A24.2k14/ASCI_A24_Day3_part2.pdf. Accessed: 17-05-2018.

[39] OpenfMRI. Openfmri. Webpage, 2018. URL https://openfmri.org/. Accessed: 02-05-2018.

Part V

Appendices

A | Naïve Implementation: Structural Similarity

Name: Group 871
Date: 19/03 - 2018

A.1 Purpose

In this test the SSIM index for the naïve implementation, see Listing 4.1, is found. A part of this test is validation of the naïve implementation. Therefore the SSIM index is both calculated with a random dictionary and with a trained dictionary.

A.2 Computer Specifications

Platform	Type
Computer	Lenovo Thinkpad E470
CPU	Intel Core i7-7500U Processor (2.70GHz 4MB)
GPU	NVIDIA® GeForce® 940MX 2 GB
RAM	8.0GB DDR4 SODIMM 2133MHz

A.3 Preliminary

In the following test multiple files are used. These files contains code for the ITKrM algorithm, a main file to run the test, file containing the training data (image data) and a file with new data for finding a sparse solution. All of these files can be found in `project_code\A`, and they are named:

- Main: combined_test.py
- ITKrM: ITKrM_seq_0.py
- OMP: OMP_fast.py
- Training data: data_batch_1
- Data compare: data_batch_test

In the file called combined_test.py different variables needs to be initialized:

Variable	Value
K	200
S	40
maxit	20
N	1024
data	data = LoadFromDataBatch.ImportData(7, 1) data = data[:40, :]
W_data	32
H_data	32
N_subpic	16

A.4 Procedure

The following lines of code should be in combined_test.py.

1. To find the SSIM index, the following lines are used:

```
from skimage.measure import compare_ssim
ssim = compare_ssim(beforeImage, afterImage)
```

2. For the random dictionary, a random dictionary matrix is generated and used in the OMP algorithm to generate a sparse representation:

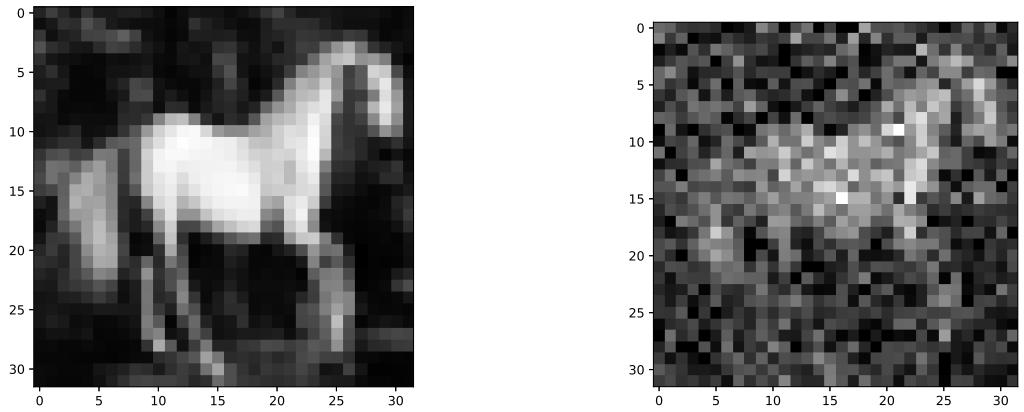
```
x_sparse = OMP_fast.OMP(D_rand, testSet[:,:N], e, S, e_or_S)
afterImage = ImportImages.MergeSmall(D_rand@x_sparse, W_data, H_data, N_subpic)
```

3. For the trained dictionary, a dictionary is trained using the ITKrM algorithm. This dictionary is then used in the OMP algorithm to generate as sparse representation:

```
dictionary = Niels_ITKrM.itkrm(smallSet,K,S,maxit)
x_sparse = OMP_fast.OMP(dictionary, testSet[:,:N], e, S, e_or_S)
afterImage = ImportImages.MergeSmall(dictionary@x_sparse, W_data, H_data, N_subpic)
```

A.5 Results

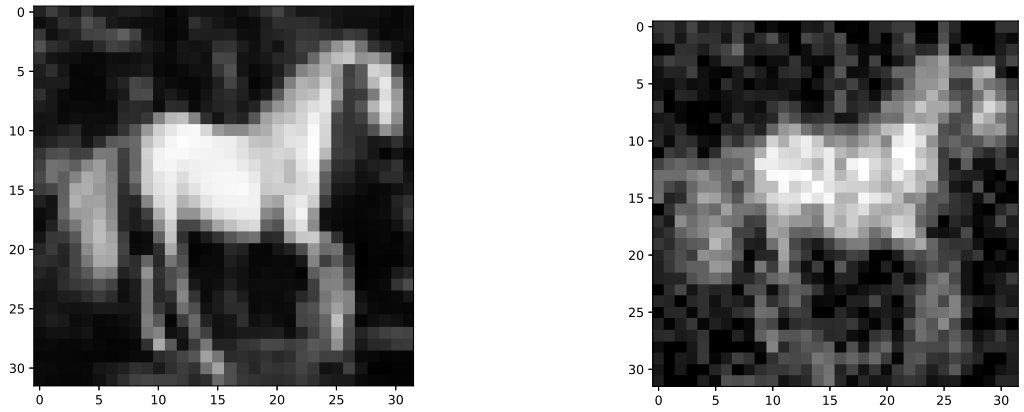
Random dictionary:



(a) beforeImage. **(b)** afterImage.

Figure A.1: Results from naïve implementation with random dictionary. SSIM index = 0.7787.

Trained dictionary:



(a) beforeImage. **(b)** afterImage.

Figure A.2: Results from naïve implementation with trained dictionary. SSIM index = 0.9212.

Appendix A. Naïve Implementation: Structural Similarity

B | ITKrM vs. DCT: Structural Similarity

Name: Group 871
Date: 17/05 - 2018

B.1 Purpose

This test is made to compare the SSIM index of the ITKrM algorithm versus that of the DCT algorithm. Furthermore the execution time of the OMP and DCT algorithms are found.

B.2 Computer Specifications

Platform	Type
Computer	Lenovo Legion Y520
CPU	Intel Core i5-7300HQ Processor (2.50GHz 6MB)
GPU	NVIDIA® GeForce® GTX 1050 Ti 4 GB
RAM	8.0GB DDR4 SODIMM 2400MHz

B.3 Preliminary

In the following test multiple files are used. These files contains code for the ITKrM algorithm, a main file to run the test and a file containing the training data (image data). All of these files can be found in `project_code\B`, and they are named:

- Main: combined_test_with_dct.py
- ITKrM: ITKrM_seq_0.py
- OMP: OMP_fast.py
- DCT: DCT_test.py
- Training data: data_batch_1

In the file called combined_test_with_dct.py different variables needs to be initialized:

Variable	Value
K	200
S	25
maxit	20
N	1024
data	data = LoadFromDataBatch.ImportData(7, 1) data = data[:40, :]
W_data	32
H_data	32
N_subpic	16

B.4 Procedure

The following lines of code should be in combined_test_with_dct.py.

1. To find SSIM index from a before and an after image, the following lines are used:

```
from skimage.measure import compare_ssim
ssim = compare_ssim(beforeImage, afterImage)
```

2. For DCT the following function is used:

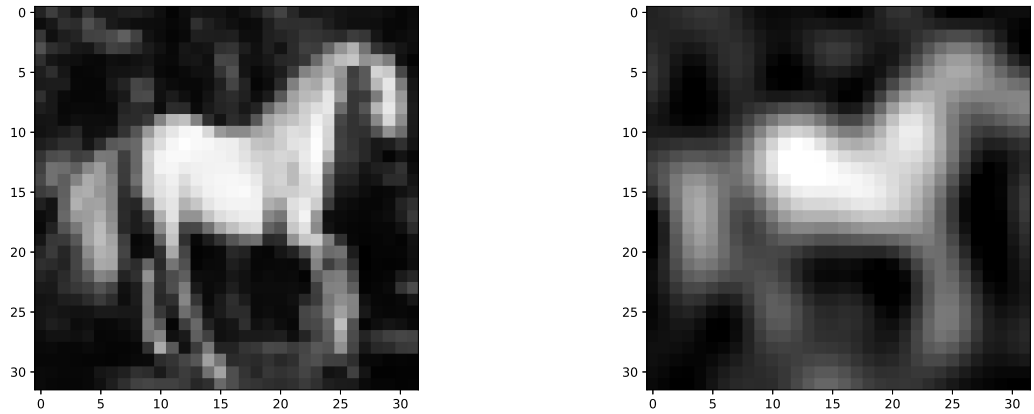
```
DCT_test.dct_picture(picture, sparsness, number of subpictures)
```

3. For the trained dictionary, a dictionary is trained using the ITKrM algorithm. This dictionary is then used in the OMP algorithm to generate as sparse representation:

```
dictionary = Niels_ITKrM.itkrm(smallSet,K,S,maxit)
x_sparse = OMP_stable.OMP(dictionary, smallSet[:,:N], e, S, e_or_S)
afterImage = ImportImages.MergeSmall(dictionary@x_sparse, W_data, H_data, N_subpic)
```

B.5 Results

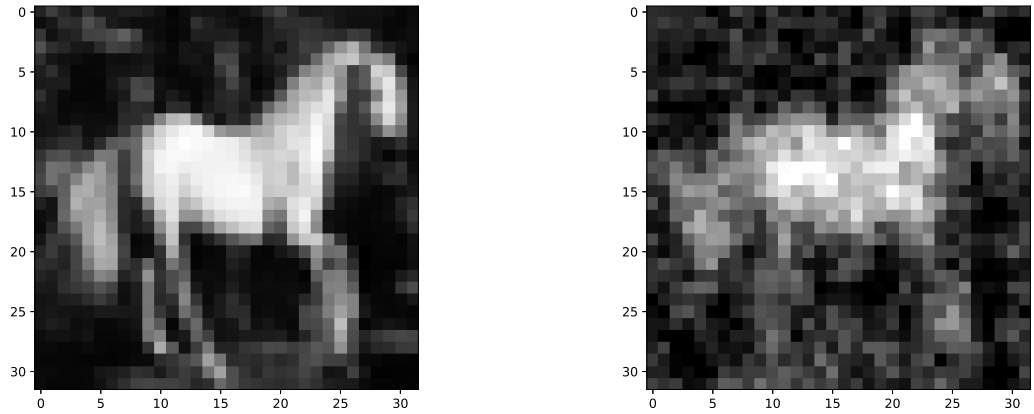
DCT:



(a) beforeImage. **(b)** afterImage.

Figure B.1: Results from DCT algorithm. SSIM index = 0.7363.

Trained dictionary:



(a) beforeImage. **(b)** afterImage.

Figure B.2: Results from naïve implementation with trained dictionary. SSIM index = 0.8984.

Log from Python terminal:

Listing B.1: Output log from running the test.

```
OMP took: 0.6204056739807129 seconds
Total ITKrM ssim: 0.8984879557809706
ITKrM non-zero entries used: 100

DCT took: 0.003299713134765625 seconds
Total DCT ssim: 0.7363479592084958
DCT non-zero entries used: 100
```

C | Sequential Optimization - Iteration 0, 1, 2 and 3

Name: Group 871
Date: 21/03 - 2018

C.1 Purpose

The following appendix shows results from four tests that finds both the total execution time and the quantity at each line in the code. The result 'iteration 0' functions as a baseline for the following three tests, iteration 1, 2 and 3. The test finds the execution time for the naïve implementation, found in Listing 4.1, for a specific used computer. The naïve implementation is then optimized and the results of the test is shown in 'iteration 1'. The code used for 'iteration 1' is then optimized again and the results are shown in 'iteration 2' and so forth. The purpose of the four tests is to find the total execution time and quantity at each line, enabling the possibility of increasing performance at each iteration.

C.2 Computer Specifications

Platform	Type
Computer	Phoenix 1 node
CPU	2 X Intel Westmere-EP architecture 4 cores (2.93 GHz)
GPU	3 X NVIDIA GTX580 GPU (3 GiB memory)
RAM	96 GiB RAM

To find the execution time the tool called line_profiler version 2.0py36_0 is used.

C.3 Preliminary

In the following test multiple files are used. These files contains code for the ITKrM algorithm, a main file to run the test and a file containing the training data (image data). All of these files can be found in `project_code\C`, and they are named:

- ITKrM: ITKrM_seq_0.py

- ITKrM: ITKrM_seq_1.py
- ITKrM: ITKrM_seq_2.py
- ITKrM: ITKrM_seq_3.py
- Main: combined_test.py
- Training data: data_batch_1

In the file called combined_test.py different variables needs to be initialized:

Variable	Value
K	200
S	40
maxit	20
N	1024
data	data = LoadFromDataBatch.ImportData(7, 1) data = data[:100, :]
W_data	32
H_data	32
N_subpic	16

C.4 Procedure

1. To find the execution time write following command in terminal:

```
kernprof -l combined_test.py
```

C.5 Results

Iteration 0:

Listing C.1: Iterative 0 of sequential optimization.

```
Timer unit: 1e-06 s

Total time: 2623.16 s
```

```
File: ITKrM_seq_0.py
Function: itkrm at line 32

Line no.    Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    32                                           @profile
    33                                           def itkrm(data,K,S,maxitr,startD=np.array
        ([1])):
    34         1            5      5.0      0.0      M, N = data.shape
    35         1           43     43.0      0.0      if startD.all()==1:
    36         1         3307   3307.0      0.0          D_init = np.random.randn(M, K)
    37                                               else:
    38                                                   D_init=startD
    39         1            1      1.0      0.0      Y = data
    40         1           17     17.0      0.0      I_D=np.zeros((S,N), dtype=np.int32)
    41
    42                                               #Algorithm
    43         1            2      2.0      0.0      D_old=D_init
    44        21           38      1.8      0.0      for i in range(maxitr):
    45        20          441     22.1      0.0          start_time=N_timer.cont_timer(0,0)
    46        20          933     46.6      0.0          N_timer.Timer(i,maxitr)
    47      8020        12748      1.6      0.0          for n in range(N):
    48      8000       284261     35.5      0.0              I_D[:,n]=max_atoms(D_old,Y[:,n],S
        )
    49        20          681     34.0      0.0          D_new=np.zeros((M,K))
    50      4020         4264      1.1      0.0          for k in range(K):
    51   1604000      2701154      1.7      0.1              for n in range(N):
    52   1600000   2136411164   1335.3     81.4                  matproj=proj(D_old[:,I_D[:,n
        ]])@Y[:,n]
    53   1600000    384668443    240.4     14.7                  vecproj=proj(D_old[:,k])@Y[:,
        n]
    54   1600000     19813538     12.4      0.8                  signer=np.sign(D_old[:,k].T@Y
        [:,n])
    55   1600000     38161970     23.9      1.5                  indicator=np.any(I_D[:,n]==k)
    56   1600000     40940790     25.6      1.6                  D_new[:,k]=D_new[:,k]+(Y[:,n
        ]-matproj+vecproj)*signer*indicator
    57
    58        20         3304    165.2      0.0          scale = np.sum(D_new*D_new, axis=0)
    59        20          403     20.1      0.0          iszero = np.where(scale < 0.00001)[0]
    60        20         7691    384.6      0.0          D_new[:,iszero] = np.random.randn(M,
        len(iszero))
    61
    62
    63        20       142313   7115.6      0.0          D_new = normalize_mat_col(D_new)
    64        20         1084     54.2      0.0          D_old = 1*D_new
    65
    66
    67         1            1      1.0      0.0      return D_old
```

Iteration 1:

Listing C.2: Iterative 1 of sequential optimization.

```
Total time: 101.303 s
File: ITKrM_seq_1.py
Function: itkrm at line 32
```

```
Line no.    Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    32                                           @profile
    33                                           def itkrm(data,K,S,maxitr,startD=np.array
        ([1])):
    34         1            3      3.0      0.0      M, N = data.shape
    35         1           30     30.0      0.0      if startD.all()==1:
    36         1         2775   2775.0      0.0          D_init = np.random.randn(M, K)
    37                                               else:
    38                                                   D_init = startD
    39         1            1      1.0      0.0      Y = data
    40         1           13     13.0      0.0      I_D = np.zeros((S,N), dtype=np.int32)
    41
    42                                               #Algorithm
    43         1            1      1.0      0.0      D_old = D_init
    44        21           40      1.9      0.0      for i in range(maxitr):
    45        20          331     16.6      0.0          start_time = N_timer.cont_timer(0,0)
    46        20          610     30.5      0.0          N_timer.Timer(i,maxitr)
    47      8020        12600      1.6      0.0          for n in range(N):
    48      8000       301474     37.7      0.3              I_D[:,n] = max_atoms(D_old,Y[:,n
        ],S)
    49        20          808     40.4      0.0          D_new = np.zeros((M,K))
    50      8020        11058      1.4      0.0          for n in range(N):
    51      8000     11026233   1378.3     10.9              matproj = proj(D_old[:,I_D[:,n]])
        @Y[:,n]
    52    328000       750731      2.3      0.7              for k in I_D[:,n]:
    53    320000     77962189    243.6     77.0                  vecproj = proj(D_old[:,k])@Y
        [:,n]
    54    320000      4115208     12.9      4.1                  signer = np.sign(D_old[:,k].
        T@Y[:,n])
    55    320000      6962337     21.8      6.9                  D_new[:,k] = D_new[:,k]+(Y[:,
        n]-matproj+vecproj)*signer
    56
    57        20         3239    161.9      0.0          scale = np.sum(D_new*D_new, axis=0)
    58        20          408     20.4      0.0          iszero = np.where(scale < 0.00001)[0]
    59        20         7396    369.8      0.0          D_new[:,iszero] = np.random.randn(M,
        len(iszero))
    60
    61
    62        20       144378   7218.9      0.1          D_new = normalize_mat_col(D_new)
    63        20         1100     55.0      0.0          D_old = 1*D_new
    64
    65
    66         1            1      1.0      0.0      return D_old
```

Iteration 2:

Listing C.3: Iterative 2 of sequential optimization.

```
Total time: 16.0257 s
File: ITKrM_seq_2.py
Function: itkrm at line 32

Line no.    Hits         Time  Per Hit   % Time  Line Contents
==============================================================
```

```
    32                                           @profile
    33                                           def itkrm(data,K,S,maxitr,startD=np.array
        ([1])):
    34         1            3      3.0      0.0      M, N = data.shape
    35         1           30     30.0      0.0      if startD.all()==1:
    36         1         2777   2777.0      0.0          D_init = np.random.randn(M, K)
    37                                               else:
    38                                                   D_init = startD
    39         1            1      1.0      0.0      Y = data
    40         1           11     11.0      0.0      I_D = np.zeros((S,N), dtype=np.int32)
    41
    42                                               #Algorithm
    43         1            1      1.0      0.0      D_old = D_init
    44        21           40      1.9      0.0      for i in range(maxitr):
    45        20          234     11.7      0.0          start_time = N_timer.cont_timer(0,0)
    46        20          444     22.2      0.0          N_timer.Timer(i,maxitr)
    47      8020        12844      1.6      0.1          for n in range(N):
    48      8000       315260     39.4      2.0              I_D[:,n] = max_atoms(D_old,Y[:,n
        ],S)
    49        20          775     38.8      0.0          D_new = np.zeros((M,K))
    50      8020        18069      2.3      0.1          for n in range(N):
    51      8000     11645781   1455.7     72.7              matproj = np.repeat(np.array([
        proj(D_old[:,I_D[:,n]])@Y[:,n] ]).T, S, axis=1)
    52      8000      1770207    221.3     11.0              vecproj = D_old[:,I_D[:,n]] @ np.
        diag(np.diag(D_old[:,I_D[:,n]].T @ D_old[:,I_D[:,n]] )**-1*(D_old[:,I_D[:,n]].T@Y[:,n
        ]))
    53      8000       322517     40.3      2.0              signer = np.sign(D_old[:,I_D[:,n
        ]].T@Y[:,n])
    54      8000      1782121    222.8     11.1              D_new[:,I_D[:,n]] = D_new[:,I_D
        [:,n]] + (np.repeat(np.array([Y[:,n]]).T, S, axis=1) - matproj + vecproj)*signer
    55
    56
    57
    58
    59
    60        20         2485    124.2      0.0          scale = np.sum(D_new*D_new, axis=0)
    61        20          343     17.1      0.0          iszero = np.where(scale < 0.00001)[0]
    62        20         7110    355.5      0.0          D_new[:,iszero] = np.random.randn(M,
        len(iszero))
    63
    64
    65        20       143634   7181.7      0.9          D_new = normalize_mat_col(D_new)
    66        20         1036     51.8      0.0          D_old = 1*D_new
    67
    68
    69         1            1      1.0      0.0      return D_old
```

Iteration 3:

Listing C.4: Iterative 3 of sequential optimization.

```
Total time: 6.38175 s
File: ITKrM_seq_3.py
Function: itkrm at line 32

Line no.    Hits         Time  Per Hit   % Time  Line Contents
```

```
============================================================
    32                                           @profile
    33                                           def itkrm(data,K,S,maxitr,startD=np.array
        ([1])):
    34         1            5      5.0      0.0      M, N = data.shape
    35         1           48     48.0      0.0      if startD.all()==1:
    36         1         2955   2955.0      0.0          D_init = np.random.randn(M, K)
    37                                               else:
    38                                                   D_init = startD
    39         1            1      1.0      0.0      Y = data
    40         1           15     15.0      0.0      I_D = np.zeros((S,N), dtype=np.int32)
    41
    42                                               #Algorithm
    43         1            1      1.0      0.0      D_old = D_init
    44        21           39      1.9      0.0      for i in range(maxitr):
    45        20          172      8.6      0.0          start_time = N_timer.cont_timer(0,0)
    46        20          374     18.7      0.0          N_timer.Timer(i,maxitr)
    47      8020        13047      1.6      0.2          for n in range(N):
    48      8000       315062     39.4      4.9              I_D[:,n] = max_atoms(D_old,Y[:,n
        ],S)
    49        20         1127     56.4      0.0          D_new = np.zeros((M,K))
    50        20         6956    347.8      0.1          DtD = D_old.T@D_old
    51      8020        18166      2.3      0.3          for n in range(N):
    52      8000       264212     33.0      4.1              DtY = D_old[:,I_D[:,n]].T @ Y[:,n
        ]
    53      8000      2587050    323.4     40.5              matproj = np.repeat(np.array([
        D_old[:,I_D[:,n]] @ np.linalg.inv(DtD[I_D[:,n,None], I_D[:,n]]) @ DtY ]).T, S, axis=1)
    54      8000      1099049    137.4     17.2              vecproj = D_old[:,I_D[:,n]] @ np.
        diag(np.diag( DtD[I_D[:,n,None], I_D[:,n]] )**-1*( DtY ))
    55      8000        48046      6.0      0.8              signer = np.sign( DtY )
    56      8000      1869248    233.7     29.3              D_new[:,I_D[:,n]] = D_new[:,I_D
        [:,n]] + (np.repeat(Y[:,n,None], S, axis=1) - matproj + vecproj)*signer
    57
    58
    59
    60
    61
    62        20         2720    136.0      0.0          scale = np.sum(D_new*D_new, axis=0)
    63        20          335     16.8      0.0          iszero = np.where(scale < 0.00001)[0]
    64        20         7018    350.9      0.1          D_new[:,iszero] = np.random.randn(M,
        len(iszero))
    65
    66
    67        20       144706   7235.3      2.3          D_new = normalize_mat_col(D_new)
    68        20         1395     69.8      0.0          D_old = 1*D_new
    69
    70
    71         1            1      1.0      0.0      return D_old
```

D | Sequential Optimization: Memory Usage

Name: Group 871
Date: 16/03 - 2018

D.1 Purpose

In this test the memory usage for the naïve implementation, see Listing 4.1, and the third iteration of the sequential optimization, see Listing C.4 are found. Here both the total memory usage is found together with the quantities at each individual line.

D.1.1 Computer Specifications

Platform	Type
Computer	Lenovo Thinkpad E470
CPU	Intel Core i7-7500U Processor (2.70GHz 4MB)
GPU	NVIDIA® GeForce® 940MX 2 GB
RAM	8.0GB DDR4 SODIMM 2133MHz

D.2 Preliminary

In the following test, multiple files are used. These files contain code for the ITKrM algorithm, a main file to run the test and a file containing the training data (image data). All of these files can be found in `project_code\D`, and they are named:

- Main: combined_test.py
- ITKrM: ITKrM_seq_0.py
- ITKrM: ITKrM_seq_3.py
- Training data: data_batch_1

In the file called combined_test.py different variables need to be initialized:

Variable	Value
K	200
S	4
maxit	1
N	1024
data	data = LoadFromDataBatch.ImportData(7, 1) data = data[:100, :]
W_data	32
H_data	32
N_subpic	16

Note, that to find the memory usage the tool memory_profiler version 0.52.0py36_0 is used.

D.3 Procedure

The following steps are used to find the memory usage of the naïve ITKrM implementation and of iteration 3 of the sequential optimization.

1. To find and save the data for memory usage write following command in terminal:
```
python -m memory_profiler combined_test.py > file_to_write_to.py.mprof.txt
```
2. Use itkrm from the file ITKrM_seq_0.py and run the command in step 1.
3. Use itkrm from the file ITKrM_seq_3.py and run the command in step 1.

D.4 Results

Naïve implementation:

Listing D.1: Memory usage in Naïve implementation of the ITKrM algorithm.

```
Line no.    Mem usage    Increment   Line Contents
================================================
    32  131.082 MiB  131.082 MiB   @profile
    33                             def itkrm(data,K,S,maxitr,startD=np.array([1])):
    34  131.082 MiB    0.000 MiB       M, N = data.shape
```

```
    35  131.086 MiB    0.004 MiB       if startD.all()==1:
    36  131.484 MiB    0.398 MiB           D_init = np.random.randn(M, K)
    37                                 else:
    38                                     D_init=startD
    39  131.484 MiB    0.000 MiB       Y = data
    40  131.488 MiB    0.004 MiB       I_D=np.zeros((S,N), dtype=np.int32)
    41                                 #Algorithm
    42  131.488 MiB    0.000 MiB       D_old=D_init
    43  134.426 MiB    0.000 MiB       for i in range(maxitr):
    44  131.488 MiB    0.000 MiB           start_time=N_timer.cont_timer(0,0)
    45  131.488 MiB    0.000 MiB           N_timer.Timer(i,maxitr)
    46  132.398 MiB    0.000 MiB           for n in range(N):
    47  132.398 MiB    0.910 MiB               I_D[:,n]=max_atoms(D_old,Y[:,n],S)
    48  132.789 MiB    0.391 MiB           D_new=np.zeros((M,K))
    49  134.215 MiB    0.000 MiB           for k in range(K):
    50  134.215 MiB    0.000 MiB               for n in range(N):
    51  134.215 MiB    1.355 MiB                   matproj=proj(D_old[:,I_D[:,n]])@Y[:,n]
    52  134.215 MiB    0.051 MiB                   vecproj=proj(D_old[:,k])@Y[:,n]
    53  134.215 MiB    0.004 MiB                   signer=np.sign(D_old[:,k].T@Y[:,n])
    54  134.215 MiB    0.008 MiB                   indicator=np.any(I_D[:,n]==k)
    55  134.215 MiB    0.008 MiB                   D_new[:,k]=D_new[:,k]+(Y[:,n]-matproj+
        vecproj)*signer*indicator
    56  134.215 MiB    0.000 MiB           scale = np.sum(D_new*D_new, axis=0)
    57  134.219 MiB    0.004 MiB           iszero = np.where(scale < 0.00001)[0]
    58  134.223 MiB    0.004 MiB           D_new[:,iszero] = np.random.randn(M, len(iszero))
    59
    60  134.426 MiB    0.203 MiB           D_new = normalize_mat_col(D_new)
    61  134.426 MiB    0.000 MiB           D_old = 1*D_new
    62  134.426 MiB    0.000 MiB       return D_old
```

Listing D.1 shows that the naïve implementation of the ITKrM algorithm uses 134.426 MiB− 131.082 MiB = 3.344 MiB.

Third sequential optimization:

Listing D.2: Memory usage in 3rd implementation of the ITKrM algorithm.

```
Line no.    Mem usage    Increment   Line Contents
================================================
    32  131.359 MiB  131.359 MiB   @profile
    33                             def itkrm(data,K,S,maxitr,startD=np.array([1])):
    34  131.359 MiB    0.000 MiB       M, N = data.shape
    35  131.363 MiB    0.004 MiB       if startD.all()==1:
    36  131.762 MiB    0.398 MiB           D_init = np.random.randn(M, K)
    37                                 else:
    38                                     D_init = startD
    39  131.762 MiB    0.000 MiB       Y = data
    40  131.762 MiB    0.000 MiB       I_D = np.zeros((S,N), dtype=np.int32)
    41                                 #Algorithm
    42  131.762 MiB    0.000 MiB       D_old = D_init
    43  135.215 MiB    0.000 MiB       for i in range(maxitr):
    44  131.762 MiB    0.000 MiB           start_time = N_timer.cont_timer(0,0)
    45  131.762 MiB    0.000 MiB           N_timer.Timer(i,maxitr)
    46  132.672 MiB    0.000 MiB           for n in range(N):
    47  132.672 MiB    0.910 MiB               I_D[:,n] = max_atoms(D_old,Y[:,n],S)
```

```
    48  133.062 MiB    0.391 MiB           D_new = np.zeros((M,K))
    49  133.961 MiB    0.898 MiB           DtD = D_old.T@D_old
    50  134.383 MiB    0.000 MiB           for n in range(N):
    51  134.383 MiB    0.000 MiB               DtY = D_old[:,I_D[:,n]].T @ Y[:,n]
    52  134.383 MiB    0.238 MiB               matproj = np.repeat(np.array([ D_old[:,I_D[:,n
        ]] @ np.linalg.inv(DtD[I_D[:,n,None], I_D[:,n]]) @ DtY ]).T, S, axis=1)
    53  134.383 MiB    0.008 MiB               vecproj = D_old[:,I_D[:,n]] @ np.diag(np.diag(
        DtD[I_D[:,n,None], I_D[:,n]] )**-1*( DtY ))
    54  134.383 MiB    0.004 MiB               signer = np.sign( DtY )
    55  134.383 MiB    0.172 MiB               D_new[:,I_D[:,n]] = D_new[:,I_D[:,n]] + (np.
        repeat(Y[:,n,None], S, axis=1) - matproj + vecproj)*signer
    56  134.777 MiB    0.395 MiB           scale = np.sum(D_new*D_new, axis=0)
    57  134.781 MiB    0.004 MiB           iszero = np.where(scale < 0.00001)[0]
    58  134.781 MiB    0.000 MiB           D_new[:,iszero] = np.random.randn(M, len(iszero))
    59
    60  135.215 MiB    0.434 MiB           D_new = normalize_mat_col(D_new)
    61  135.215 MiB    0.000 MiB           D_old = 1*D_new
    62  135.215 MiB    0.000 MiB       return D_old
```

Listing D.2 shows that the 3. iteration of the sequential optimization of the ITKrM algorithm uses $135.215\,\mathrm{MiB} - 131.359\,\mathrm{MiB} = 3.856\,\mathrm{MiB}$.

E | Parallel CPU Optimization - Iteration 1

Name: Group 871
Date: 18/04 - 2018

E.1 Purpose

In this test three aspects of the first parallel optimization iteration on a CPU will be investigated. The three aspects are:

- Execution time
- Speed-up potential
- Amount of overhead

E.2 Computer Specifications

Platform	Type
Computer	Lenovo Legion Y520
CPU	Intel Core i5-7300HQ Processor (2.50GHz 6MB)
GPU	NVIDIA® GeForce® GTX 1050 Ti 4 GB
RAM	8.0GB DDR4 SODIMM 2400MHz

E.3 Preliminary

In the following test multiple files are used. These files contains code for the ITKrM algorithm, a main file to run the test and a file containing the training data (image data). All of these files can be found in `project_code\E`, and they are named:

- Main: combined_test_par_cpu.py
- ITKrM: ITKrM_parallel_map.py

- Training data: data_batch_1

In the file called combined_test_par_cpu.py different variables needs to be initialized:

Variable	Value
K	200
S	40
maxit	20
N	1024
nTrainingData	100
data	data = LoadFromDataBatch.ImportData(-1, 1) data = data[:nTrainingData, :]
W_data	32
H_data	32
N_subpic	16

E.4 Procedure

Execution time:
The following steps are made to find the execution time:

1. Allow the program to run the parallel part of the program using one process.
2. Set t0 = time.time() in the beginning of the ITKrM algorithm.
3. Calculate executionTime = time.time()-t0 in the end of the ITKrM algorithm.
4. Repeat step 2 and 3 with nTrainingData = $[10, 100, 500, 1000, 1500, 2000, 3000, 4000]$.
5. Repeat step 2, 3 and 4 with using 4 processes.

Speed-up potential:
To find the speed-up potential the following steps are made:

1. Allow the program to run the parallel part of the program using only one process.

2. Set t0 = time.time() and t_par = 0 in the beginning of the ITKrM algorithm.
3. Set t_par_0 = time.time() right before the parallel part of the program executes.
4. calculate t_par = t_par + time.time()-t_par_0 right after the parallel part of the program has executed to find the parallel execution time. t_par is added to take the iterations of the ITKrM algorithm into account.
5. Calculate t_seq = time.time()-t0-t_par in the end of the ITKrM algorithm to find the sequential execution time.
6. Calculate $t_{ratio} = \frac{\text{t\_par}}{\text{t\_seq}}$ for nTrainingData = $[10, 100, 500, 1000, 1500, 2000, 3000, 4000, 6000]$ and decide, based on the results, whether Amdahl's or Gustafson-Barsis' law is fitting.
7. Calculate the execution time of the entire program using one process as t_exe = t_seq + t_par

Amount of overhead:
The following steps are made to find the amount of overhead when using multiprocessing:

1. Allow the program to run the parallel part of the program using only one process.
2. Calculate the parallel execution time t_par in the same way as in the procedure for the speed-up potential.
3. Repeat step 2 using two and four processes.
4. Calculate the amount of overhead as $T_{oh} = T_{par\_x} - \frac{T_{par\_1}}{x}$, where T_{par_x} is the parallel execution time using x processes.
5. Repeat step 1, 2, 3 and 4 for nTrainingData = $[10, 100, 500, 1000, 1500, 2000, 3000, 4000]$

E.5 Results

Execution time:
The results of the execution time test is presented in Figure E.1.

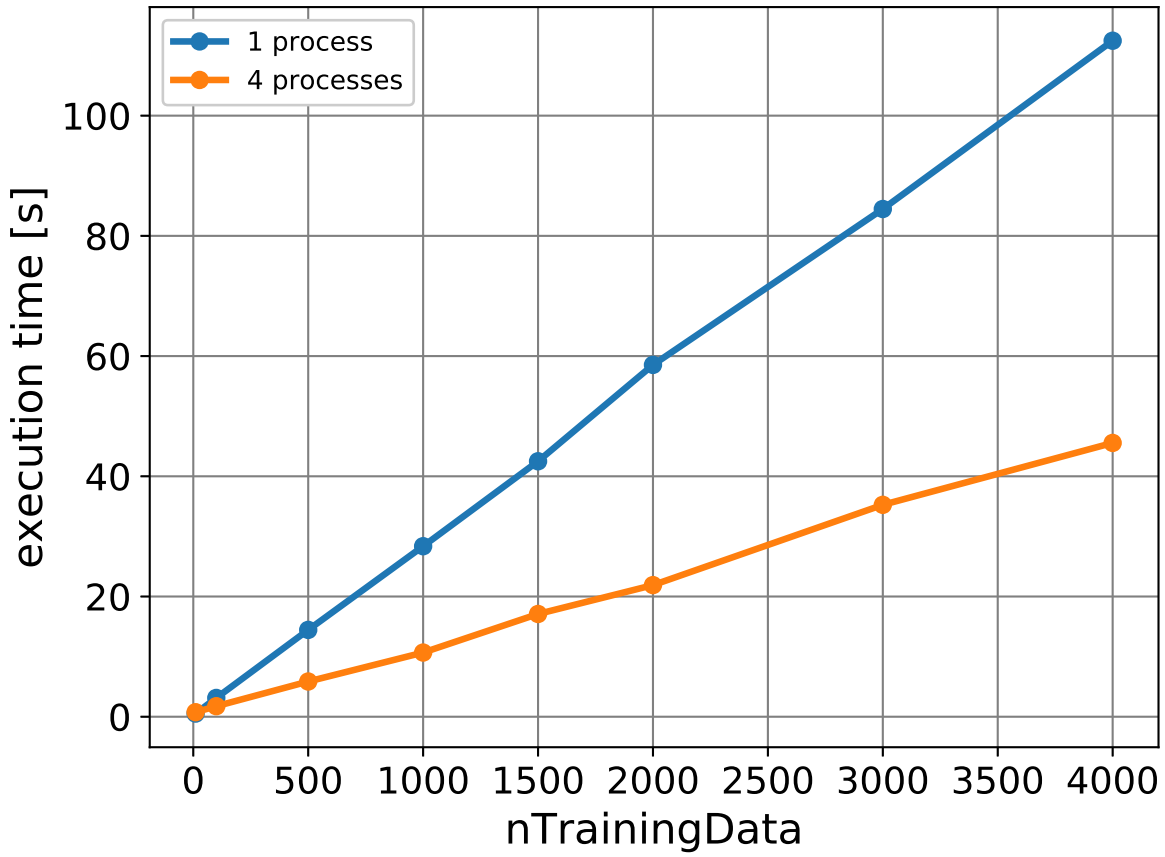


Figure E.1: Execution using 1 and 4 processes.

Speed-up potential

The results of the speed-up potential test is presented in Table E.1 and Figure E.2.

Table E.1: `t_seq` and `t_exe` speed-up potential results.

nTrainingData	**10**	**100**	**500**	**1000**	**1500**	**2000**	**3000**	**4000**	**6000**
t_seq [s]	0.10	0.23	0.87	1.63	2.44	3.33	5.07	6.66	9.39
t_exe	0.50	3.03	14.97	29.21	43.60	58.42	87.97	116.38	175.61

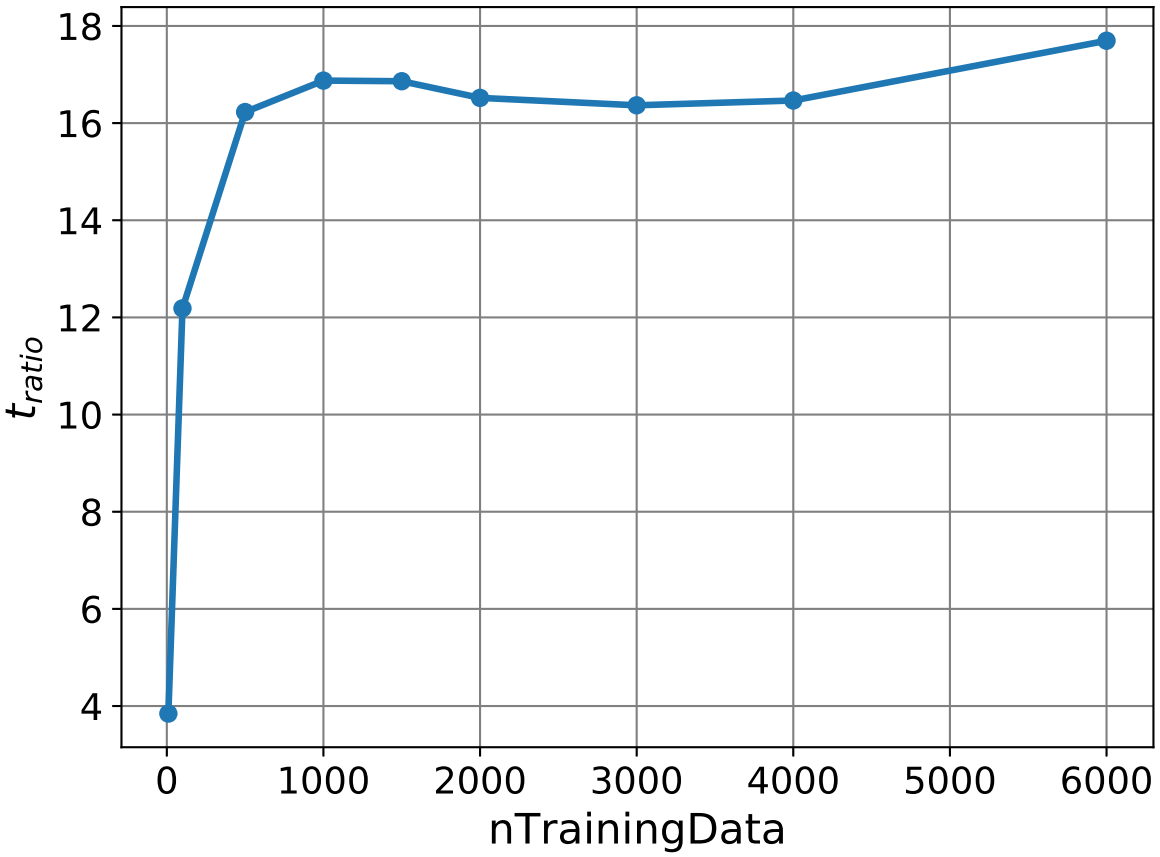


Figure E.2: Graph of the ratio between the parallel- and the sequential- execution time.

Amount of overhead

The results of the amount of overhead test is presented in Figure E.3.

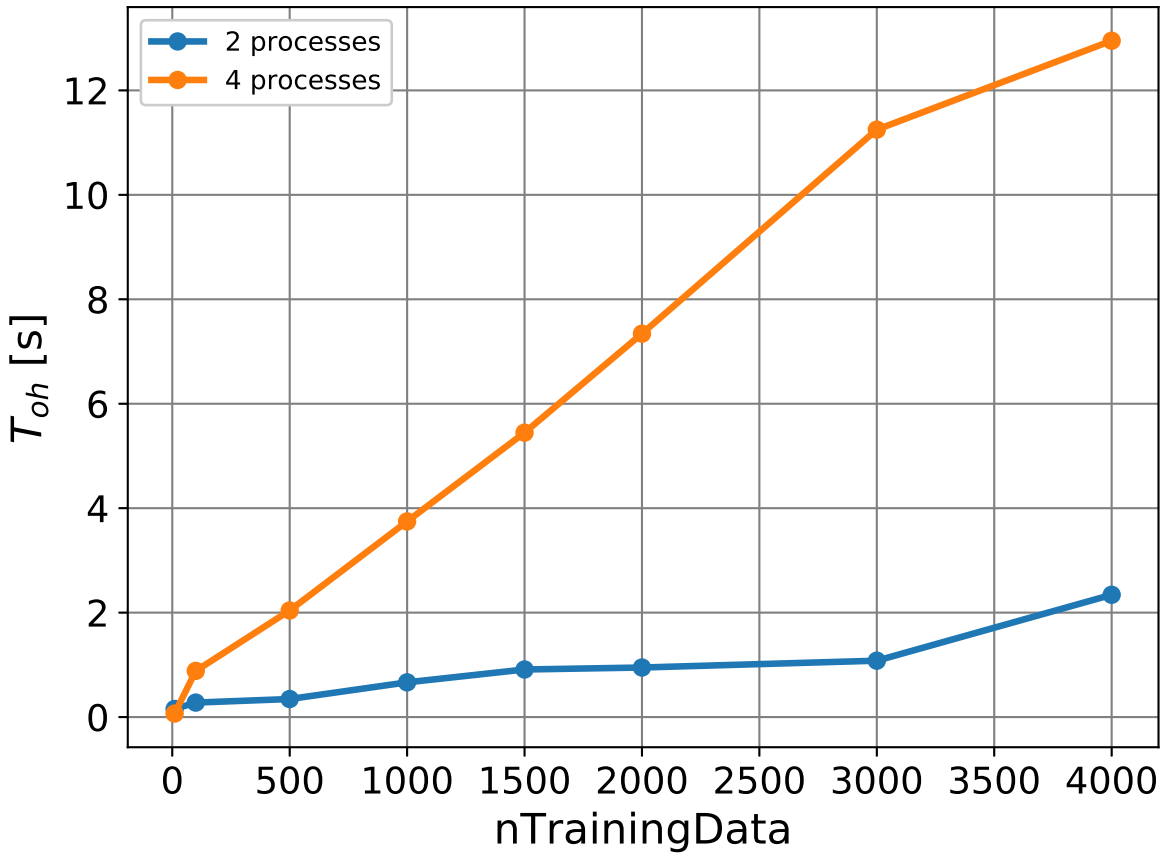


Figure E.3: Amount of overhead using 2 and 4 processes.

F | Speed-up using Multiprocessing

Name: Group 871
Date: 16/05 - 2018

F.1 Purpose

A test is made to investigate how the use of multiple processes on a CPU affects the execution of the ITKrM implementation. The speed-up is tested on two devices, on the Phoenix 1 node in the Birdnest cluster and a reference PC.

F.2 Computer Specifications

Phoenix 1 node:

Platform	Type
Computer	Phoenix 1 node
CPU	2 X Intel Westmere-EP architecture 4 cores (2.93 GHz)
GPU	3 X NVIDIA GTX580 GPU (3 GiB memory)
RAM	96 GiB RAM

Reference PC:

Platform	Type
Computer	Custom
CPU	AMD Ryzen 7 1700 (3.0GHz 16MB)
GPU	NVIDIA® GeForce® GTX 1060 6 GB
RAM	16.0GB DDR4 DIMM 2400MHz

F.3 Preliminary

In the following test multiple files are used. These files contains code for the ITKrM algorithm, a main file to run the test and a file containing the training data (image data). All of these files can be found in `project_code\F`, and they are named:

- Main: combined_test_apply_2.py
- ITKrM: ITKrM_parallel_map.py
- Training data: data_batch_1

In the file called combined_test_apply_2.py different variables needs to be initialized:

Variable	Value
K	200
S	40
maxit	20
N	1024
nTrainingData	1000
data	data = LoadFromDataBatch.ImportData(-1, 1) data = data[:nTrainingData, :]
W_data	32
H_data	32
N_subpic	16

F.4 Procedure

The speed-up test is made using the following steps on both the Phoenix node and on the reference PC:

1. Run the file combined_test_apply_2.py when allowing the ITKrM algorithm to run in 1 process.
2. Note the execution time.
3. Repeat step 1 and 2 with the number of processes = $[2, 4, 8]$.

F.5 Results

Speed-up on Phoenix node
The result of the speed-up test on the Phoenix node is seen in Figure F.1.

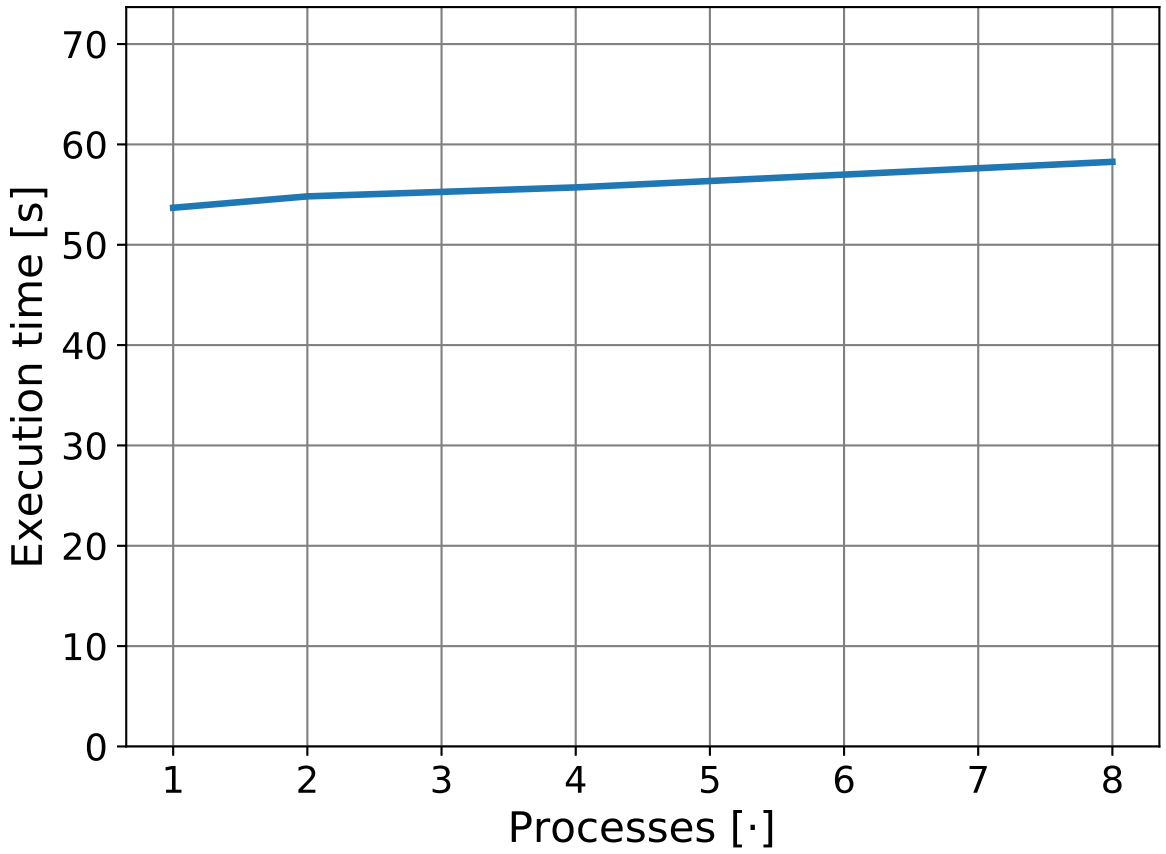


Figure F.1: Shows the execution for increasing number of processes on Phoenix node.

Speed-up on reference PC

The result of the speed-up test on the reference PC is seen in Figure F.2.

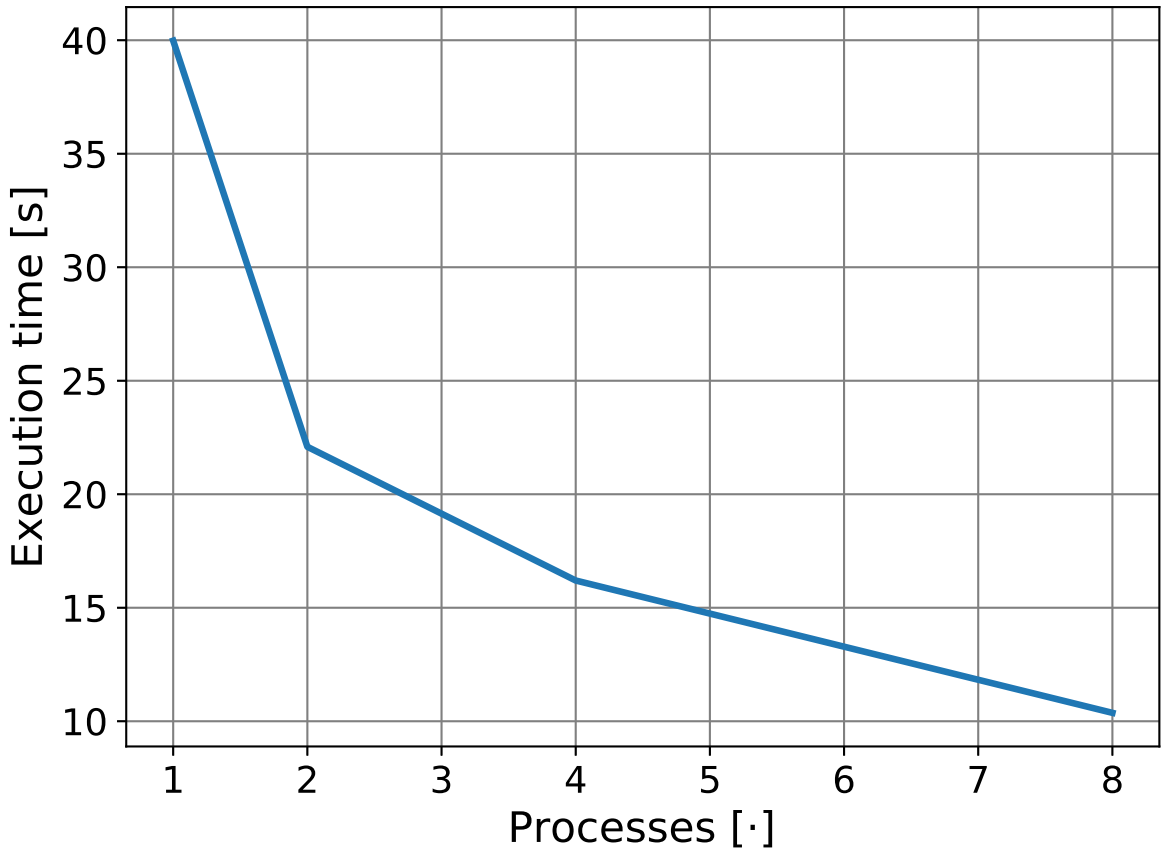


Figure F.2: Shows the execution for increasing number of processes on reference PC.

G | GPU Preliminaries

Name: Group 871
Date: 10/05 - 2018

G.1 Purpose

In this test some preliminary aspects of the GPU are tested. What is tested is:

- Square matrix multiplication execution time GPU vs CPU
- Matrix inversion execution time GPU vs CPU

G.2 Computer Specifications

Platform	Type
Computer	Lenovo Legion Y520
CPU	Intel Core i5-7300HQ Processor (2.50GHz 6MB)
GPU	NVIDIA® GeForce® GTX 1050 Ti 4 GB
RAM	8.0GB DDR4 SODIMM 2400MHz

G.3 Preliminary

In the following test multiple files are used. All of these files can be found in `project_code\G`, and they are named:

- mat_mul_testing.py
- matrix_inversion_test.py

G.4 Procedure

Square matrix multiplication execution time:
The following steps are made to get the execution times:

1. Run mat_mul_testing.py

Matrix inversion execution time:
The following steps are made to get the execution times:

1. Run matrix_inversion_test.py

G.5 Results

Square matrix multiplication execution time:
The results of the execution time test is presented as a figure in Figure G.1, and the raw data in Table G.1.

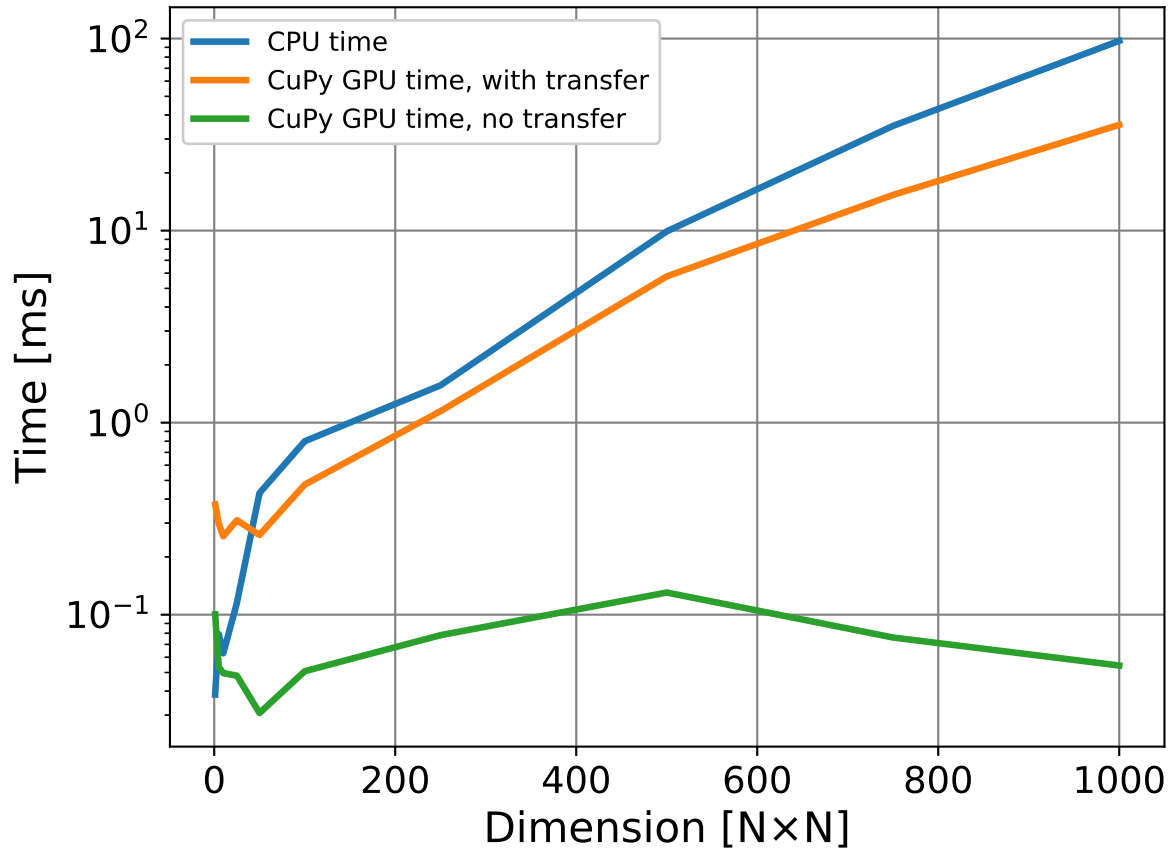


Figure G.1: Shows the execution time on GPU vs. CPU for a square matrix multiplication.

Table G.1: Shows the execution times on GPU vs. CPU for a square matrix multiplication.

Dimension [N×N]	**CPU [ms]**	**GPU with transfer [ms]**	**GPU no transfer [ms]**
1	0.0383853912	0.3750324249	0.1003742218
5	0.0791549683	0.2985000610	0.0534057617
10	0.0629425049	0.2558231354	0.0495910645
25	0.1151561737	0.3104209900	0.0481605530
50	0.4298686981	0.2601146698	0.0307559967
100	0.8003711700	0.4758834839	0.0507831573
250	1.5640258789	1.1472702026	0.0782012939
500	9.9093914032	5.7756900787	0.1304149628
750	35.0959300995	15.3341293335	0.0760555267
1000	97.2881317139	35.5432033539	0.0543594360

Matrix inversion execution time:
The results of the execution time test is presented in Figure G.2, and the raw data in Table G.2.

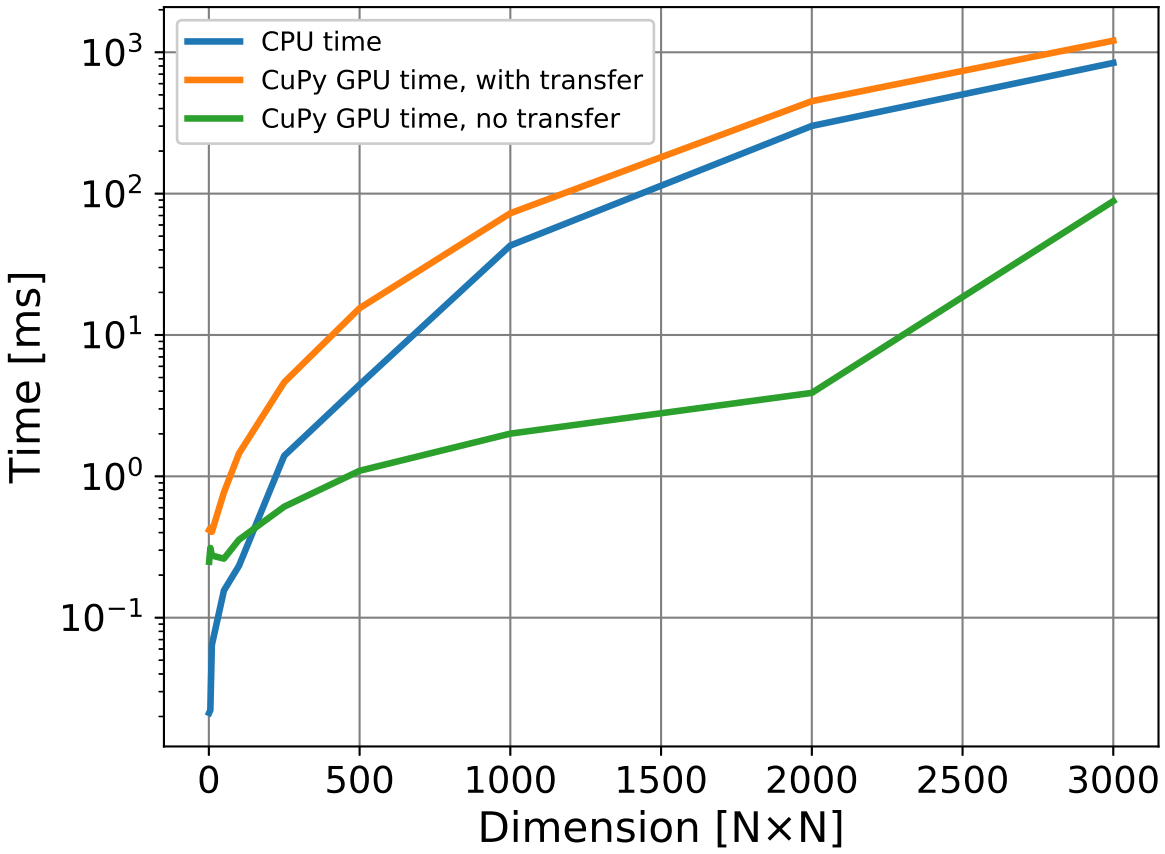


Figure G.2: Shows the execution time on GPU vs. CPU for a matrix inversion.

Table G.2: Shows the execution times on GPU vs. CPU for a matrix inversion.

Dimension [N×N]	**CPU [ms]**	**GPU with transfer [ms]**	**GPU no transfer [ms]**
1	0.0212192535	0.4169940948	0.2486705780
5	0.0221729279	0.4312992096	0.3113746643
10	0.0641345978	0.4045963287	0.2751350403
50	0.1556873322	0.7696151733	0.2615451813
100	0.2329349518	1.4533996582	0.3561973572
250	1.3978481293	4.6281814575	0.6098747253
500	4.4500827789	15.4690742493	1.0948181152
1000	42.9503917694	72.6752281189	2.0046234131
2000	301.2015819550	450.9317874908	3.8936138153
3000	841.1145210266	1207.7598571777	88.5503292084

H | Parallel GPU Optimization - Naïve

Name: Group 871
Date: 9/05 - 2018

H.1 Purpose

In this test 4 aspects of the naïve GPU implementation of ITKrM will be investigated. The 4 aspects are:

- Validation
- Line profiling
- Execution time
- Time used on data transfer

H.2 Computer Specifications

Platform	Type
Computer	Lenovo Legion Y520
CPU	Intel Core i5-7300HQ Processor (2.50GHz 6MB)
GPU	NVIDIA® GeForce® GTX 1050 Ti 4 GB
RAM	8.0GB DDR4 SODIMM 2400MHz

H.3 Preliminary

In the following test multiple files are used. These files contains code for the ITKrM algorithm, a main file to run the test and a file containing the training data (image data). All of these files can be found in `project_code\H`, and they are named:

- Main: combined_test.py
- ITKrM: ITKrM_GPU_0.py and ITKrM_seq_3.py
- Training data: data_batch_1

The reason for using two ITKrM algorithms is to ensure that the time used for comparison is found on the same hardware.
In the file called combined_test.py different variables needs to be initialized:

Variable	Value
K	200
S	40
maxit	20
N	1024
data	data = LoadFromDataBatch.ImportData(-1, 1) data = data[:nTrainingData, :]
W_data	32
H_data	32
N_subpic	16

H.4 Procedure

Validation:
The following steps are made to validate the implementation:

1. Set nTrainingData to 100
2. Set ITKrM to GPU_itkrm.itkrm
3. In the file GPU_itkrm.py make sure the line for CPU to generate random numbers is commented in, and the line for GPU to generate random is commented out.
4. Run combined_test.py
5. Save the variable dictionary
6. Set ITKrM to CPU_itkrm.itkrm
7. Run combined_test.py
8. Compare the new variable dictionary with the old dictionary variable using np.allclose()

Line profiling:
The following steps are made to do a line profiling:

1. Set nTrainingData to 100
2. Set ITKrM to GPU_itkrm.itkrm
3. Run

```
kernprof -l combined_test.py
```

4. Save the data using

```
python -m line_profiler combined_test.py.lprof > GPU_line_profiler_100.txt
```

Execution time:
The variable nTrainingData should be changed, as to get a grasp of the influence of the amount of traning data. nTrainingData $= [10, 100, 500, 1000, 1500, 2000, 3000, 4000]$. The following steps are made to find the execution time:

1. Set nTrainingData
2. Set ITKrM to GPU_itkrm.itkrm
3. In the file GPU_itkrm.py make sure the line for CPU to generate random numbers is commented out, and the line for GPU to generate random is commented in.
4. Run combined_test.py
5. Save the execution time
6. Set ITKrM to CPU_itkrm.itkrm
7. Run combined_test.py
8. Save the execution time
9. Repeat for each entry in nTrainingData

Time used on data transfer:
To time used on data transfer the command nvprof is utilized. The steps are as follows:

1. Set nTrainingData to 100
2. Set ITKrM to GPU_itkrm.itkrm
3. Run in a terminal

```
nvprof python combined_test.py
```

4. Copy the results from the terminal into at file.

H.5 Results

Validation:
The np.allclose() commmand returned true, which means the dictionary that is generated with the GPU implementation is the same as the one using the CPU implementation.
Line profiling:
The results of the line profiling is presented in Listing H.1.

Listing H.1: Line profile of naïve GPU implementation.

```
Timer unit: 1e-06 s

Total time: 20.8742 s
File: ITKrM_GPU_0.py
Function: itkrm at line 27

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    27                                           @profile
    28                                           def itkrm(data,K,S,maxitr,startD=np.array
        ([1])):
    29         1          3.0      3.0      0.0      M, N = data.shape
    30         1         61.0     61.0      0.0      if startD.all()==1:
    31         1       1707.0   1707.0      0.0          D_init = np.random.randn(M, K)
    32                                               else:
    33                                                   D_init = startD
    34
    35                                               #Algorithm
    36         1      10445.0  10445.0      0.1      GPU_D_old = cp.asarray(D_init)
    37
    38         1        192.0    192.0      0.0      GPU_Y = cp.asarray(data)
    39
    40         1     124072.0 124072.0      0.6      GPU_M = int(cp.asarray(M))
    41
    42         1        104.0    104.0      0.0      GPU_N = int(cp.asarray(N))
    43
    44         1         67.0     67.0      0.0      GPU_S = int(cp.asarray(S))
    45
    46         1         63.0     63.0      0.0      GPU_maxitr = int(cp.asarray(maxitr))
    47
    48         1        236.0    236.0      0.0      GPU_I_D = cp.zeros((S,N),dtype=cp.int32)
    49
    50        21         25.0      1.2      0.0      for i in range(GPU_maxitr):
    51        20        142.0      7.1      0.0          start_time = N_timer.cont_timer(0,0)
    52        20        538.0     26.9      0.0          N_timer.Timer(i,maxitr)
    53      8020      10859.0      1.4      0.1          for n in range(GPU_N):
    54      8000    2630746.0    328.8     12.6              GPU_I_D[:,n] = max_atoms(
        GPU_D_old,GPU_Y[:,n],GPU_S)
    55
    56        20        426.0     21.3      0.0          GPU_D_new = cp.zeros((M,K))
    57
    58        20       2518.0    125.9      0.0          GPU_DtD = GPU_D_old.T @ GPU_D_old
    59
    60      8020      11744.0      1.5      0.1          for n in range(GPU_N):
    61      8000    1830114.0    228.8      8.8              GPU_DtY = GPU_D_old[:,GPU_I_D[:,n
        ]].T @ GPU_Y[:,n]
    62      8000    7438928.0    929.9     35.6              GPU_matproj = cp.repeat((
        GPU_D_old[:,GPU_I_D[:,n]] @ cp.linalg.inv(GPU_DtD[GPU_I_D[:,n,None], GPU_I_D[:,n]]) @
        GPU_DtY)[:,None],GPU_S,axis=1)
    63      8000    5541899.0    692.7     26.5              GPU_vecproj = GPU_D_old[:,GPU_I_D
        [:,n]] @ cp.diag(cp.diag( GPU_DtD[GPU_I_D[:,n,None], GPU_I_D[:,n]] )**-1*( GPU_DtY ))
    64      8000     270757.0     33.8      1.3              GPU_signer = cp.sign( GPU_DtY )
    65      8000    2403796.0    300.5     11.5              GPU_D_new[:,GPU_I_D[:,n]] =
        GPU_D_new[:,GPU_I_D[:,n]] + (cp.repeat(GPU_Y[:,n,None], S, axis=1) - GPU_matproj +
        GPU_vecproj)*GPU_signer
```

```
66
67
68
69
70        20       2625.0    131.2      0.0          GPU_scale = cp.sum(GPU_D_new*
    GPU_D_new, axis=0)
71        20       6952.0    347.6      0.0          GPU_iszero = cp.where(GPU_scale <
    0.00001)[0]
72
73        20       8680.0    434.0      0.0          GPU_D_new[:,GPU_iszero] = cp.asarray(
    np.random.randn(M, len(GPU_iszero)))  # generate random with CPU
74        20     574585.0  28729.2      2.8          GPU_D_new = normalize_mat_col(
    GPU_D_new)
75        20       1646.0     82.3      0.0          GPU_D_old = 1*GPU_D_new
76         1        223.0    223.0      0.0      return cp.asnumpy(GPU_D_old)
```

Execution time:
The results of the execution time test is presented in Figure H.1, and the raw data in Table H.1.

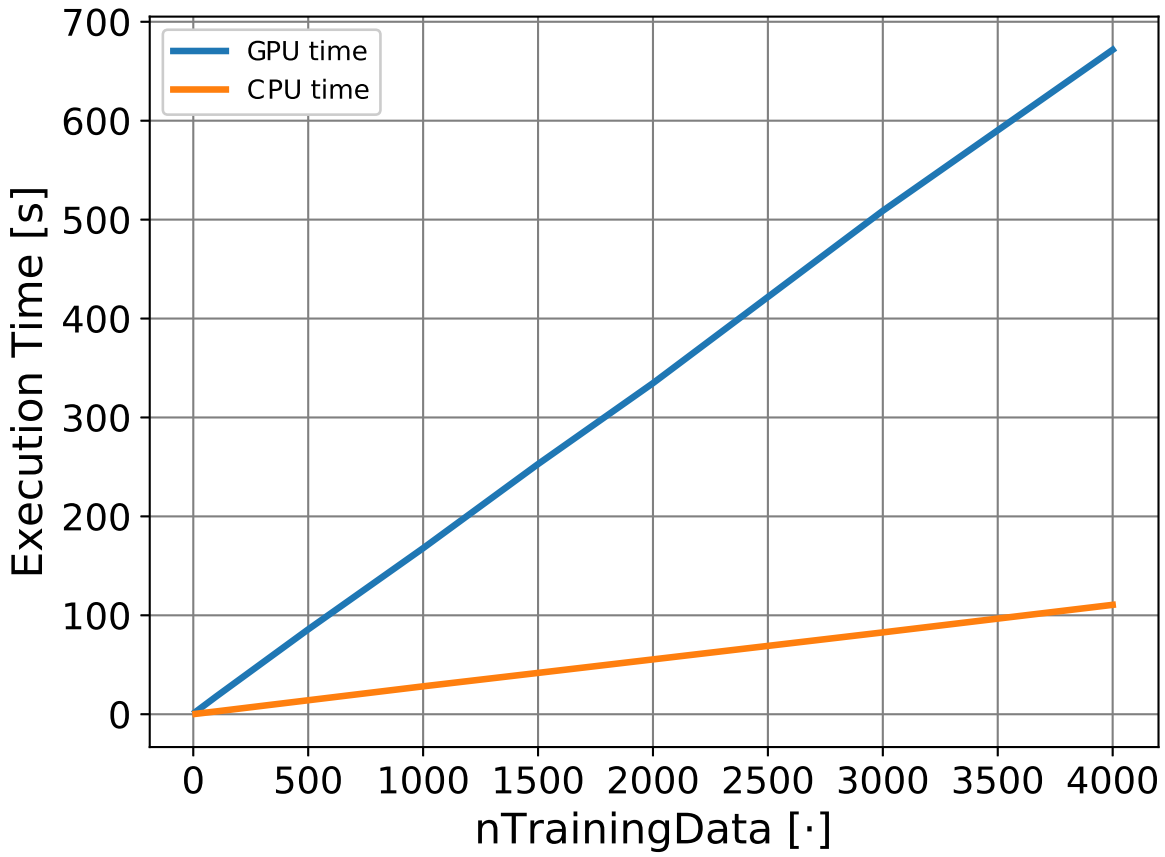


Figure H.1: Shows the execution times found through testing.

Table H.1: Shows the execution times found through testing

nTrainingData	GPU [seconds]	CPU [seconds]
10	2.36907339096069	0.391240835189819
100	18.0006823539734	2.84894847869873
500	85.7833416461945	14.1323614120483
1000	167.863515853882	28.1977562904358
1500	252.875040531158	41.7826945781708
2000	334.66859292984	55.4318046569824
3000	508.892999887466	82.732319355011
4000	671.636948108673	110.589530706406

Time used on data transfer

The results of nvprof is presented in Listing H.2.

Listing H.2: nvprof of naïve GPU implementation.

```
==9547== Profiling application: python combined_test.py
==9547== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   41.32%  1.81137s     18477  98.033us  51.041us  124.42us  void
     getf2_domino_kernel<double, double, int=256, int=7, bool=1>(int, int, double*, int, int*,
      int, int*, int*, double*, int*, double*)
                   22.15%  970.80ms     37211  26.089us  5.3760us  443.37us  void
                       gemm_kernel2x2_core<double, bool=0, bool=0, bool=0, bool=0, bool=0>(
                       double*, double const *, double const *, int, int, int, int, int, int,
                       double*, double*, double, double, int)
                    6.16%  270.21ms      6159  43.872us  43.712us  44.289us  void
                        trsm_left_kernel<double, int=256, int=4, bool=0, bool=0, bool=0, bool
                        =1, bool=0>(cublasTrsmParams<double>, double, double const *, int)
                    5.18%  227.19ms     93386  2.4320us  1.0240us  16.481us  cupy_copy
                    4.20%  184.26ms     36955  4.9860us  2.2400us  16.064us  cupy_take
                    3.58%  156.82ms      6400  24.502us  23.969us  139.52us  void thrust::
                        cuda_cub::cub::DeviceRadixSortSingleTileKernel<thrust::cuda_cub::cub::
                        DeviceRadixSortPolicy<double, unsigned long, int>::Policy700, bool=0,
                        double, unsigned long, int>(int const *, thrust::cuda_cub::cub::
                        DeviceRadixSortSingleTileKernel<thrust::cuda_cub::cub::
                        DeviceRadixSortPolicy<double, unsigned long, int>::Policy700, bool=0,
                        double, unsigned long, int>*, thrust::cuda_cub::cub::
                        DeviceRadixSortPolicy<double, unsigned long, int>::Policy700 const *,
                        thrust::cuda_cub::cub::DeviceRadixSortSingleTileKernel<thrust::
                        cuda_cub::cub::DeviceRadixSortPolicy<double, unsigned long, int>::
                        Policy700, bool=0, double, unsigned long, int>**, bool=0, int, int)
                    3.00%  131.50ms     30795  4.2700us  2.3040us  16.161us  void
                        laswp_kernel2<double, bool=0>(int, double*, int, int, int, int const
                        *, int)
                    2.27%  99.538ms     12318  8.0800us  4.3520us  16.225us  void
                        trsm_left_kernel<double, int=256, int=4, bool=0, bool=0, bool=0, bool
```

```
                    =0, bool=1>(cublasTrsmParams<double>, double, double const *, int)
12                1.73%  75.665ms      6159  12.285us  12.129us  15.905us  void
                    trsm_ln_kernel<double, unsigned int=32, unsigned int=32, unsigned int
                    =4, bool=1>(int, int, double const *, int, double*, int, double,
                    double const *, int, int*)
13                1.57%  69.034ms     36984  1.8660us  1.0240us  16.256us  cupy_multiply
14                1.26%  55.242ms     24636  2.2420us  1.4720us  16.160us  cupy_remainder
15                1.02%  44.802ms     15333  2.9210us  2.3040us  18.112us  cupy_sum
16                1.01%  44.352ms      6174  7.1830us  4.3520us  15.584us
                    cupy_scatter_update
17                1.00%  43.647ms      9159  4.7650us  4.0960us  15.872us  cupy_power
18                0.75%  32.664ms     43388     752ns     480ns  16.033us  [CUDA memset]
19                0.63%  27.488ms      6159  4.4630us  4.3200us  16.160us  void
                    gemmSN_NN_kernel<double, double, double, int=128, int=2, int=4, int=8,
                     int=4, int=4>(cublasGemmSmallNParams<double, double, double>, double
                    const *, double const *, double, double, int)
20                0.55%  23.891ms     12318  1.9390us  1.7280us  16.128us  cupy_add
21                0.38%  16.588ms      6159  2.6930us  2.3040us  15.905us  cupy_subtract
22                0.37%  16.061ms     12318  1.3030us  1.0240us  15.936us  copy_info_kernel(
                    int, int*)
23                0.25%  11.011ms      6159  1.7870us  1.5680us  15.425us  cupy_sign
24                0.24%  10.547ms      6159  1.7120us  1.6320us  15.617us  getf2_init(int*)
25                0.21%  9.2438ms      6400  1.4440us  1.1520us  15.872us  cupy_absolute
26                0.20%  8.9813ms      6159  1.4580us  1.3440us  15.712us  dtrsv_init(int*)
27                0.19%  8.2517ms      6159  1.3390us  1.2480us  15.809us
                    pivotOffset_kernel(int*, int, int)
28                0.19%  8.1898ms      6400  1.2790us  1.0880us  15.584us  void thrust::
                    cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::
                    ParallelForAgent<thrust::cuda_cub::__transform::unary_transform_f<
                    double*, double*, thrust::cuda_cub::__transform::no_stencil_tag,
                    thrust::identity<double>, thrust::cuda_cub::__transform::
                    always_true_predicate>, long>, thrust::cuda_cub::__transform::
                    unary_transform_f<double*, double*, thrust::cuda_cub::__transform::
                    no_stencil_tag, thrust::identity<double>, thrust::cuda_cub::
                    __transform::always_true_predicate>, long>(double*, thrust::cuda_cub::
                    __transform::no_stencil_tag)
29                0.18%  7.7042ms      6400  1.2030us  1.0880us  15.937us  void thrust::
                    cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::
                    ParallelForAgent<thrust::cuda_cub::__transform::binary_transform_f<
                    thrust::counting_iterator<unsigned long, thrust::use_default, thrust::
                    use_default, thrust::use_default>, thrust::constant_iterator<long,
                    thrust::use_default, thrust::use_default>, thrust::device_ptr<unsigned
                     long>, thrust::cuda_cub::__transform::no_stencil_tag, thrust::modulus
                    <unsigned long>, thrust::cuda_cub::__transform::always_true_predicate
                    >, __int64>, thrust::cuda_cub::__transform::binary_transform_f<thrust
                    ::counting_iterator<unsigned long, thrust::use_default, thrust::
                    use_default, thrust::use_default>, thrust::constant_iterator<long,
                    thrust::use_default, thrust::use_default>, thrust::device_ptr<unsigned
                     long>, thrust::cuda_cub::__transform::no_stencil_tag, thrust::modulus
                    <unsigned long>, thrust::cuda_cub::__transform::always_true_predicate
                    >, __int64>(thrust::use_default, thrust::use_default)
30                0.18%  7.6741ms      6400  1.1990us  1.0880us  15.552us  void thrust::
                    cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::
                    ParallelForAgent<thrust::cuda_cub::__transform::unary_transform_f<
                    unsigned long*, unsigned long*, thrust::cuda_cub::__transform::
                    no_stencil_tag, thrust::identity<unsigned long>, thrust::cuda_cub::
```

```
                        __transform::always_true_predicate>, long>, thrust::cuda_cub::
                        __transform::unary_transform_f<unsigned long*, unsigned long*, thrust
                        ::cuda_cub::__transform::no_stencil_tag, thrust::identity<unsigned
                        long>, thrust::cuda_cub::__transform::always_true_predicate>, long>(
                        unsigned long*, thrust::cuda_cub::__transform::no_stencil_tag)
31                  0.12%  5.2119ms      3000  1.7370us  1.5040us  15.264us  cupy_sqrt
32                  0.11%  4.7680ms      3000  1.5890us  1.5360us  15.584us  cupy_true_divide
33                  0.01%  419.79us        37  11.345us     544ns  184.77us  [CUDA memcpy HtoD
                        ]
34                  0.00%  168.42us        15  11.228us  10.944us  11.584us
                        cupy_concatenate_same_size
35                  0.00%  63.329us        15  4.2210us  3.8400us  4.3520us
                        inclusive_scan_kernel
36                  0.00%  30.817us        15  2.0540us  1.4720us  9.4410us  cupy_not_equal
37                  0.00%  22.304us        15  1.4860us  1.4720us  1.5360us  nonzero_1d_kernel
38                  0.00%  20.096us        15  1.3390us  1.2800us  1.4080us  cupy_less
39                  0.00%  16.832us        19     885ns     704ns  1.0560us  [CUDA memcpy DtoH
                        ]
```

I | LU Decomposition for Solving a System of Linear Equations

The LU decomposition can be used to solve a system of linear equations $\boldsymbol{Ax} = \boldsymbol{b}$, by decomposing $\boldsymbol{A}$ [1]. The LU decomposition of the square matrix $\boldsymbol{A}$ is,

$$\boldsymbol{A} = \boldsymbol{LU} \tag{I.1}$$

where

$$\boldsymbol{L} = \begin{bmatrix} l_{0,0} & & & \\ l_{1,0} & l_{1,1} & & \\ \vdots & \vdots & \ddots & \\ l_{i,0} & l_{i,1} & \dots & l_{i,i} \end{bmatrix}, \quad \boldsymbol{U} = \begin{bmatrix} u_{0,0} & u_{0,1} & \dots & u_{0,i} \\ & u_{1,1} & \dots & u_{1,i} \\ & & \ddots & \vdots \\ & & & u_{i,i} \end{bmatrix} \tag{I.2}$$

So $\boldsymbol{L}$ is a lower triangular matrix and $\boldsymbol{U}$ an upper triangular matrix. The solution to the system can then be written as seen in Equation (I.3).

$$\boldsymbol{LUx} = \boldsymbol{b} \tag{I.3}$$

By setting $\boldsymbol{Ux} = \boldsymbol{z}$ the solution to the system of linear equations can be found by first solving Equation (I.4a) and then solving Equation (I.4b).

$$\boldsymbol{Lz} = \boldsymbol{b} \tag{I.4a}$$

$$\boldsymbol{Ux} = \boldsymbol{z} \tag{I.4b}$$

This might seem like more work, but as $\boldsymbol{L}$ and $\boldsymbol{U}$ are triangular matrices, the systems can easily be solved by use of forward and backwards substitution. This means that each equation reduces the next equation in the system to an equation with one unknown, by use of all previous found answers. Forward and backwards simply refers to the fact that when solving lower triangular matrix you start from the top and substitute the answer found forward to the next row, as in going from row 1 forward to row 2, and vice versa for an upper triangular matrix. It is not that easy to decompose a matrix into an upper and a lower triangular matrix, and herein the work lies when using the LU decomposition to solve a system of linear equations. In the next section, one of the possible ways to do it is shown.

I.1 Crout decomposition

The Crout decomposition[1] is found by first making a column of the lower triangular matrix then a row of the upper triangular matrix and so forth. It is defined as shown in

[1] https://www.gamedev.net/articles/programming/math-and-physics/matrix-inversion-using-lu-decomposition-r3637/

Equation (I.5) for a square N by N matrix.

$$\text{For } j = 0, ..., N-1$$

$$l_{i,j} = a_{i,j} - \sum_{k=0}^{j-1} l_{i,k} \cdot u_{k,j}, \text{ for } i = j, ..., N-1 \tag{I.5a}$$

$$u_{j,k} = \frac{a_{j,k} - \sum_{i=0}^{j-1} l_{j,i} \cdot u_{i,k}}{l_{j,j}}, \text{ for } k = j+1, ..., N-1 \tag{I.5b}$$

It should be noted that the diagonal of the upper triangular is predefined to be ones. This is also why the loop of finding the upper triangular matrix iterates over k for $j+1$, such that the diagonal is never calculated. This can be implemented in code. As the use of LU decomposition in this project is with regard to running it on a GPU, it has been chosen to write the code with matrices unrolled into vectors. The implementation of the Crout decomposition is shown in Listing I.1.

Listing I.1: An implementation of Crout decomposition.

```
1  for i in range(0, K):
2    for j in range(i, K):
3        L[i + j*K] = A[i + j*K]
4        for k in range(0, i):
5            L[i + j*K] -= L[k + j*K] * U[i + k*K]

7    U[i + i*K] = 1
8    for j in range(i+1, K):
9        U[j + i*K] = A[j + i*K] / L[i + i*K]
10       for k in range(0, i):
11           U[j + i*K] -= (L[k + i*K] * U[j + k*K]) / L[i + i*K]
```

In Listing I.1 the lines 2 to 5 finds the elements of i-th column of the lower triangular matrix, and then from lines 7 to 11 the i-th row of the upper triangular matrix is found. This is then looped for all i columns of the lower triangular matrix and all i rows of the upper triangular matrix. Now that it is possible to find the LU decomposition, an implementation to do the forward and backwards substitution is made.

I.2 Solve Linear System of Equations Implementation

It is wanted to use the found LU decomposition to solve a system of linear equations. This is done through forward and backwards substitution to solve Equation (I.4a) and Equation (I.4b). Same as before the matrices are unrolled to vectors. The implementation is shown in Listing I.2.

Listing I.2: An implementation of forward and backward substition to solve linear equation system.

```
1  for i in range(K):
2    z[i] = b[i] / L[i + i*K]
3    for j in range(0, i):
4        z[i] -= (L[j + i*K] * z[j]) / L[i + i*K]
5  for i in range(K-1, 0-1, -1):
6    x[i] = z[i]
```

```
7   for j in range(i+1, K):
8       x[i] -= U[j + i*K] * x[j]
```

What can be seen in Listing I.2 is that from line 1 to 4 the i-th row of the system of equations in Equation (I.4a) is solved to find $z[i]$, then that $z[i]$ is used to solve the next row to find $z[i+1]$ and so forth, going from the top towards the bottom of the lower triangular system of equations. Then in line 5 to 8 the rows of the system of equations in Equation (I.4b) is solved to find x, much as before with using prior found x entries to find the next entry in x, going from the bottom towards the top of the upper triangular system of equations.

J | Evaluation of Custom CUDA Kernel for Matrix and Vector Projections

Name: Group 871
Date: 15/05 - 2018

J.1 Purpose

This test investigates four aspects of the custom CUDA kernel implementation of the ITKrM algorithm. These aspects are:

- Validation of correctness
- Line profiling
- Execution time
- Time used on data transfer

J.2 Computer Specifications

Platform	Type
Computer	Custom
CPU	AMD Ryzen 7 1700 (3.0GHz 16MB)
GPU	NVIDIA® GeForce® GTX 1060 6 GB
RAM	16.0GB DDR4 DIMM 2400MHz

J.3 Preliminary

In order to validate correctness the result from the kernel implementation needs to be compared to a previous implementation. All of the needed files can be found in `project_code\J`, and they are named:

- Main and ITKrM: GPU_itkrm_1.py
- ITKrM: ITKrM_seq_3.py
- Training data: data_batch_1

The values for different variables in GPU_itkrm_1.py are initialized as follows:

Variable	Value
K	200
S	40
maxit	20
N	1024
data	data = LoadFromDataBatch.ImportData(-1, 1) data = data[:nTrainingData, :]
W_data	32
H_data	32
N_subpic	16

Throughout the testing nTrainingData will be set to $[100, 1000, 4000]$.

J.4 Procedure

The procedure to test the four aspects of the kernel implementation are as follows.

J.4.1 Validation

1. Set nTrainingData to 100
2. Set ITKrM = CPU_itkrm.itkrm
3. Run GPU_itkrm_1.py
4. Save the resulting dictionary in new variable
5. Set ITKrM = gpu_itkrm
6. Run GPU_itkrm_1.py
7. Compare the resulting dictionary with previous dictionary using np.allclose()

J.4.2 Line Profiling

1. Set nTrainingData to 100
2. Set ITKrM = gpu_itkrm
3. Run

```
kernprof -l GPU_itkrm_1.py
```

4. Save the data using

```
python -m line_profiler GPU_itkrm_1.py.lprof > kernel_0_100.txt
```

5. Repeat with nTrainingData set to 1000 and 4000

J.4.3 Execution time

The execution time is noted in the result of the Line Profiling. As the other tests regarding execution time for the sequentially optimized implementation was not done on this computer, the execution time of the 3rd iteration of sequentially optimized implementation is run with nTrainingData set to 100, 1000 and 4000 and the execution times are noted.

J.4.4 Time used on data transfer

1. Set nTrainingData to 1000
2. In a terminal on a pc with NVIDIA toolkit installed, run

```
nvprof python GPU_itkrm_1.py
```

3. Copy results from the terminal into a file

J.5 Results

J.5.1 Validation

The np.allclose() function returned true, which means the dictionaries are identical aside from small numerical inaccuracies and the implementation is thus validated for correctness.

J.5.2 Line Profiling

The results of the line profiling for nTrainingData set to 1000 is shown in Listing J.1.

Listing J.1: Line profile of kernel implementation with nTrainingData set to 1000.

```
Timer unit: 2.77277e-07 s

Total time: 28.5959 s
File: GPU_itkrm_1.py
Function: gpu_itkrm at line 20

Line No.      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    20                                           @profile
    21                                           def gpu_itkrm(data, K, S, maxit):
    22         1         15.0     15.0      0.0      M, N = data.shape
    23         1       5398.0   5398.0      0.0      D_init = np.random.randn(M, K)
    24       201        723.0      3.6      0.0      for i in range(K):
    25       200      16711.0     83.6      0.0          D_init[:,i] = D_init[:,i] / np.linalg
        .norm(D_init[:,i], 2)
    26         1          3.0      3.0      0.0      Y = data
```

```
    27         1        904.0    904.0      0.0      I_D = np.zeros((S, N), dtype=np.int32)
    28         1          3.0      3.0      0.0      D = D_init
    29         1          4.0      4.0      0.0      TpB = 32    # ThreadsPerBlock
    30         1         17.0     17.0      0.0      BpG = math.ceil(N/32)     # BlocksPerGrid
    31        21         85.0      4.0      0.0      for t in range(maxit):
    32        20       9887.0    494.4      0.0          N_timer.Timer(t, maxit)
    33     80020     299309.0      3.7      0.3          for n in range(N):
    34     80000    7672312.0     95.9      7.4              I_D[:,n] = np.argpartition(np.abs
        (D.T@Y[:,n]), -S)[-S:]
    35        20       3971.0    198.6      0.0          D_new = np.zeros((M, K))
    36        20      42767.0   2138.3      0.0          DtD = D.T@D
    37        20      11425.0    571.2      0.0          d_D = D.reshape(-1)*1
    38        20       2021.0    101.0      0.0          d_DtD = DtD.reshape(-1)*1
    39        20       9618.0    480.9      0.0          d_I_D = I_D.reshape(-1)*1
    40        20       9462.0    473.1      0.0          d_DtY = np.zeros(N*S)
    41        20     301445.0  15072.2      0.3          d_Y = Y.reshape(-1)*1
    42        20    2378558.0 118927.9      2.3          vecproj = np.zeros(N*M*S)
    43        20     394397.0  19719.8      0.4          L = np.zeros(N*S*S)
    44        20     385793.0  19289.7      0.4          U = np.zeros(N*S*S)
    45        20      12061.0    603.0      0.0          z = np.zeros(N*S)
    46        20      11785.0    589.2      0.0          x = np.zeros(N*S)
    47        20      64024.0   3201.2      0.1          matproj = np.zeros(M*N)
    48        20   40710155.0 2035507.8     39.5           D_kernel2[TpB, BpG](M, N, K, S, d_D,
         d_DtD, d_I_D, d_DtY, d_Y, vecproj, L, U, z, x, matproj)
    49     80020     355044.0      4.4      0.3          for n in range(N):
    50     80000    4084230.0     51.1      4.0              DtY = D[:,I_D[:,n]].T @ Y[:,n]
    51     80000     719691.0      9.0      0.7              signer = np.sign( DtY )
    52     80000   45319192.0    566.5     43.9              D_new[:,I_D[:,n]] = D_new[:,I_D
        [:,n]] + (np.repeat(Y[:,n,None], S, axis=1) - np.repeat(matproj.reshape(N, M)[n,:,None
        ], S, axis=1) + vecproj.reshape(N,M,S)[n,:,:])*signer
    53
    54        20       6322.0    316.1      0.0          scale = np.sum(D_new*D_new, axis=0)
    55        20        815.0     40.8      0.0          iszero = np.where(scale < 0.00001)[0]
    56        20       3453.0    172.7      0.0          D_new[:,iszero] = np.random.randn(M,
        len(iszero))
    57
    58        20     297669.0  14883.5      0.3          D_new = normalize_mat_col(D_new)
    59        20       1956.0     97.8      0.0          D = 1*D_new
    60         1          3.0      3.0      0.0      return D
```

The line profiling of nTrainingData set to 100 and 4000 are shown in Listing J.2 and Listing J.3 respectively. They only show lines 48-52 as it is specifically the time of the GPU part that is of interest.

Listing J.2: Line profile of kernel implementation with nTrainingData set to 100.

```
Timer unit: 2.77277e-07 s

Total time: 4.53588 s
File: GPU_itkrm_1.py
Function: gpu_itkrm at line 20

Line No.      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    48        20    9630902.0 481545.1     58.9          D_kernel2[TpB, BpG](M, N, K, S, d_D,
        d_DtD, d_I_D, d_DtY, d_Y, vecproj, L, U, z, x, matproj)
```

```
    49      8020      37888.0      4.7      0.2          for n in range(N):
    50      8000     425917.0     53.2      2.6              DtY = D[:,I_D[:,n]].T @ Y[:,n]
    51      8000      75873.0      9.5      0.5              signer = np.sign( DtY )
    52      8000    4594224.0    574.3     28.1              D_new[:,I_D[:,n]] = D_new[:,I_D
        [:,n]] + (np.repeat(Y[:,n,None], S, axis=1) - np.repeat(matproj.reshape(N, M)[n,:,None
        ], S, axis=1) + vecproj.reshape(N,M,S)[n,:,:])*signer
```

Listing J.3: Line profile of kernel implementation with nTrainingData set to 4000.

```
Timer unit: 2.77277e-07 s

Total time: 110.89 s
File: GPU_itkrm_1.py
Function: gpu_itkrm at line 20

Line No.      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    48        20  157360907.0 7868045.3     39.3          D_kernel2[TpB, BpG](M, N, K, S, d_D,
         d_DtD, d_I_D, d_DtY, d_Y, vecproj, L, U, z, x, matproj)
    49    320020    1435527.0      4.5      0.4          for n in range(N):
    50    320000   16379740.0     51.2      4.1              DtY = D[:,I_D[:,n]].T @ Y[:,n]
    51    320000    2954102.0      9.2      0.7              signer = np.sign( DtY )
    52    320000  178572555.0    558.0     44.7              D_new[:,I_D[:,n]] = D_new[:,I_D
        [:,n]] + (np.repeat(Y[:,n,None], S, axis=1) - np.repeat(matproj.reshape(N, M)[n,:,None
        ], S, axis=1) + vecproj.reshape(N,M,S)[n,:,:])*signer
```

It is seen that with nTrainingData set to 100, the custom kernel takes up 58.9 % of the time. With it set to 1000 and 4000 it is only 39.5 % and 39.3 % respectively.

J.5.3 Execution Time

The execution time of the kernel implementation and the 3rd iteration of sequential optimization are shown in Table J.1 and Figure J.1

Table J.1: Shows the execution times found through testing.

nTrainingData	GPU [seconds]	CPU [seconds]
100	4.54	3.89
1000	28.60	37.65
4000	110.89	149.91

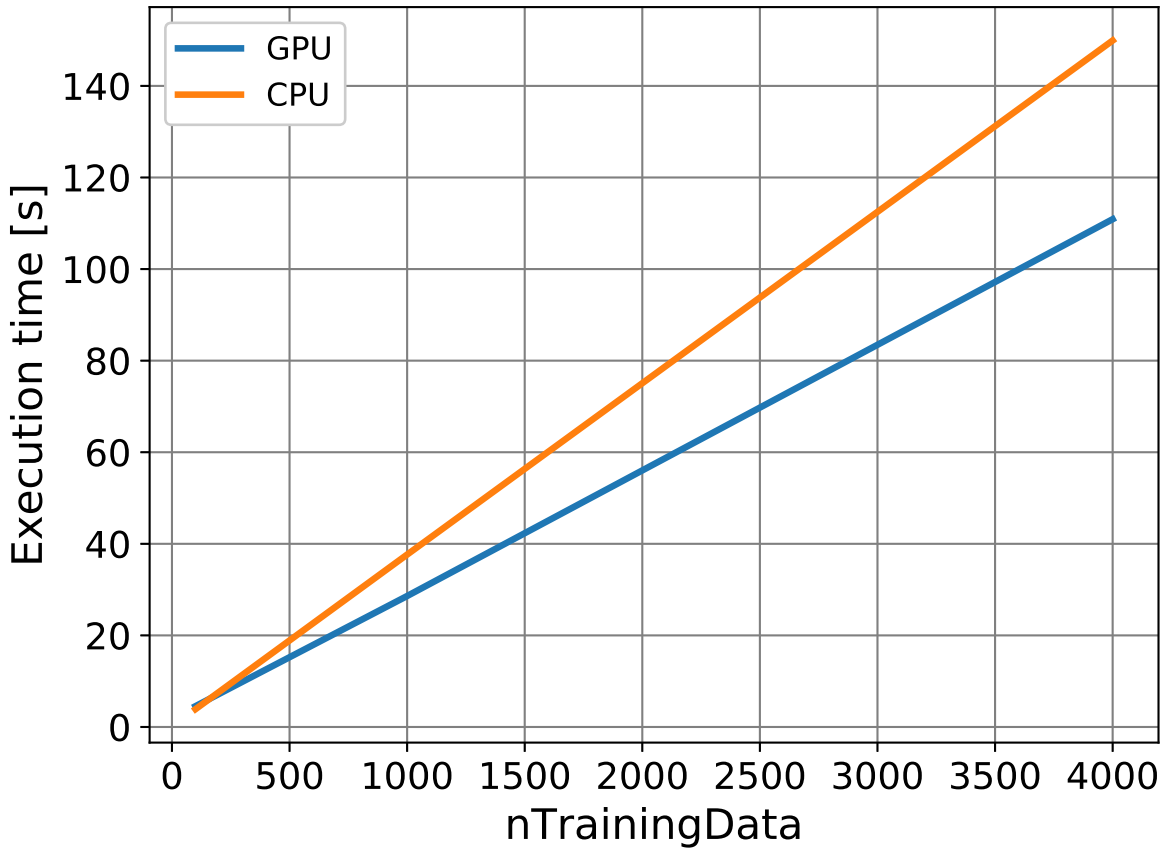


Figure J.1: Execution time of the kernel implementation compared with that of the sequentially optimized implementation.

J.5.4 Time used on data transfer

The results of nvprof is shown in Listing J.4

Listing J.4: nvprof of custom kernel for matrix and vector projections

```
1 ==9324== Profiling application: python gpu_itkrm.py
2 ==9324== Profiling result:
3 Time(%)      Time     Calls       Avg       Min       Max  Name
4  45.62%  4.38219s        20  219.11ms  215.03ms  220.00ms  cudapy::__main__::D_kernel2$241(
      __int64, __int64, __int64, __int64, Array<double, int=1, C, mutable, aligned>, Array<
      double, int=1, C, mutable, aligned>, Array<int, int=1, C, mutable, aligned>, Array<double
      , int=1, C, mutable, aligned>, Array<double, int=1, C, mutable, aligned>, Array<double,
      int=1, C, mutable, aligned>, Array<double, int=1, C, mutable, aligned>, Array<double, int
      =1, C, mutable, aligned>, Array<double, int=1, C, mutable, aligned>, Array<double, int=1,
       C, mutable, aligned>, Array<double, int=1, C, mutable, aligned>)
5  40.30%  3.87058s       220  17.594ms  25.375us  137.14ms  [CUDA memcpy HtoD]
6  14.08%  1.35206s       220  6.1457ms  24.671us  53.292ms  [CUDA memcpy DtoH]
```

K | Evaluation of Custom CUDA Kernels for ITKrM

Name: Group 871
Date: 17/05 - 2018

K.1 Purpose

This test investigates four aspects of the custom CUDA kernel implementation of the ITKrM algorithm. These aspects are:

- Validation of correctness
- Line profiling
- Execution time
- Time used on data transfer

K.2 Computer Specifications

Platform	Type
Computer	Custom
CPU	AMD Ryzen 7 1700 (3.0GHz 16MB)
GPU	NVIDIA® GeForce® GTX 1060 6 GB
RAM	16.0GB DDR4 DIMM 2400MHz

K.3 Preliminary

In order to validate correctness the result from the kernel implementation needs to be compared to a previous implementation. All of the needed files can be found in `project_code\K`, and they are named:

- Main and ITKrM: GPU_itkrm_2.py
- ITKrM: ITKrM_seq_3.py
- Training data: data_batch_1

The values for different variables in GPU_itkrm_2.py are initialized as follows:

Variable	Value
K	200
S	40
maxit	20
N	1024
data	data = LoadFromDataBatch.ImportData(-1, 1) data = data[:nTrainingData, :]
W_data	32
H_data	32
N_subpic	16

Throughout the testing nTrainingData will be set to $[100, 1000, 4000]$.

K.4 Procedure

The procedure to test the 4 aspects of the kernel implementation are as follows.

K.4.1 Validation

1. Set nTrainingData to 100
2. Set ITKrM = CPU_itkrm.itkrm
3. Run GPU_itkrm_2.py
4. Save the resulting dictionary in new variable
5. Set ITKrM = gpu_itkrm
6. Run GPU_itkrm_2.py
7. Compare the resulting dictionary with previous dictionary using np.allclose()

K.4.2 Line Profiling

1. Set nTrainingData to 100
2. Set ITKrM = gpu_itkrm
3. Run

```
kernprof -l GPU_itkrm_2.py
```

4. Save the data using

```
python -m line_profiler GPU_itkrm_2.py.lprof > kernel_0_100.txt
```

5. Repeat with nTrainingData set to 1000 and 4000

K.4.3 Execution time

The execution time is noted in the result of the Line Profiling. The execution times of the previous kernel implementation and the sequential implementation are found in Appendix J and Appendix C respectively.

K.4.4 Time used on data transfer

1. Set nTrainingData to 1000
2. In a terminal on a pc with NVIDIA toolkit installed, run

```
nvprof python GPU_itkrm_2.py
```

3. Copy results from the terminal into a file

K.5 Results

K.5.1 Validation

The np.allclose() function returned true, which means the dictionaries are identical aside from small numerical inaccuracies and the implementation is thus validated for correctness.

K.5.2 Line Profiling

The results of the line profiling for nTrainingData set to 1000 is shown in Listing K.1.

Listing K.1: Line profile of 2nd kernel implementation with nTrainingData set to 1000.

```
Timer unit: 2.77277e-07 s

Total time: 6.37819 s
File: GPU_itkrm_2.py
Function: gpu_itkrm at line 20

Line No.    Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    20                                           @profile
    21                                           def gpu_itkrm(data, K, S, maxit):
    22         1         15.0     15.0      0.0      M, N = data.shape
    23         1       5307.0   5307.0      0.0      D_init = np.random.randn(M, K)
    24       201        843.0      4.2      0.0      for i in range(K):
    25       200      16851.0     84.3      0.1          D_init[:,i] = D_init[:,i] / np.linalg
        .norm(D_init[:,i], 2)
    26         1          3.0      3.0      0.0      Y = data
    27         1        701.0    701.0      0.0      I_D = np.zeros((S, N), dtype=np.int32)
    28         1          4.0      4.0      0.0      D = D_init
```

```
    29         1          3.0      3.0      0.0      TpB = 32    # ThreadsPerBlock
    30         1         18.0     18.0      0.0      BpG_N = math.ceil(N/32)     #
       BlocksPerGrid
    31         1          5.0      5.0      0.0      BpG_M = math.ceil(M/32)
    32
    33                                               # Move training data to device and
       allocate arrays on device.
    34         1      19690.0  19690.0      0.1      d_Y = cuda.to_device(Y.reshape(-1)*1)
    35         1      40358.0  40358.0      0.2      d_vecproj = cuda.device_array(shape=(N*M*
       S))
    36         1       4025.0   4025.0      0.0      z = cuda.device_array(shape=(N*S))
    37         1       1500.0   1500.0      0.0      x = cuda.device_array(shape=(N*S))
    38         1       2421.0   2421.0      0.0      d_matproj = cuda.device_array(shape=(M*N)
       )
    39         1       1543.0   1543.0      0.0      d_signer = cuda.device_array(shape=(S*N))
    40
    41        21        104.0      5.0      0.0      for t in range(maxit):
    42        20       6637.0    331.9      0.0          N_timer.Timer(t, maxit)
    43     80020     315831.0      3.9      1.4          for n in range(N):
    44     80000    7857017.0     98.2     34.2              I_D[:,n] = np.argpartition(np.abs
       (D.T@Y[:,n]), -S)[-S:]
    45        20       3647.0    182.3      0.0          D_new = np.zeros((M, K))
    46        20      47490.0   2374.5      0.2          DtD = D.T@D
    47
    48        20      55607.0   2780.3      0.2          d_D = cuda.to_device(D.reshape(-1)*1)
    49        20      34892.0   1744.6      0.2          d_DtD = cuda.to_device(DtD.reshape
       (-1)*1)
    50        20      44053.0   2202.7      0.2          d_I_D = cuda.to_device(I_D.reshape
       (-1)*1)
    51        20      32629.0   1631.5      0.1          d_Dnew = cuda.to_device(D_new.reshape
       (-1)*1)
    52
    53        20    4261751.0 213087.5     18.5          k_matvecproj[TpB, BpG_N](d_D, d_DtD,
       d_I_D, d_Y, d_vecproj, z, x, d_matproj, d_signer)
    54
    55        20    1144033.0  57201.7      5.0          k_updateD[TpB, BpG_M](d_Dnew, d_I_D,
       d_Y, d_vecproj, d_matproj, d_signer)
    56        20    8782310.0 439115.5     38.2          D_new = d_Dnew.copy_to_host()
    57        20        386.0     19.3      0.0          D_new = D_new.reshape(M, K)
    58
    59        20       7473.0    373.6      0.0          scale = np.sum(D_new*D_new, axis=0)
    60        20        903.0     45.1      0.0          iszero = np.where(scale < 0.00001)[0]
    61        20       4193.0    209.7      0.0          D_new[:,iszero] = np.random.randn(M,
       len(iszero))
    62
    63        20     309105.0  15455.2      1.3          D_new = normalize_mat_col(D_new)
    64        20       1607.0     80.3      0.0          D = 1*D_new
    65         1          4.0      4.0      0.0      return D
```

The line profiling of nTrainingData set to 100 and 4000 are shown in Listing K.2 and Listing K.3 respectively.

Listing K.2: Line profile of 2nd kernel implementation with nTrainingData set to 100.

```
Timer unit: 2.77277e-07 s

Total time: 2.39685 s
```

```
File: GPU_itkrm_2.py
Function: gpu_itkrm at line 20

Line No.    Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    20                                           @profile
    21                                           def gpu_itkrm(data, K, S, maxit):
    22         1         15.0     15.0      0.0      M, N = data.shape
    23         1       5353.0   5353.0      0.1      D_init = np.random.randn(M, K)
    24       201        836.0      4.2      0.0      for i in range(K):
    25       200      16893.0     84.5      0.2          D_init[:,i] = D_init[:,i] / np.linalg
        .norm(D_init[:,i], 2)
    26         1          3.0      3.0      0.0      Y = data
    27         1         98.0     98.0      0.0      I_D = np.zeros((S, N), dtype=np.int32)
    28         1          3.0      3.0      0.0      D = D_init
    29         1          3.0      3.0      0.0      TpB = 32    # ThreadsPerBlock
    30         1         16.0     16.0      0.0      BpG_N = math.ceil(N/32)     #
        BlocksPerGrid
    31         1          4.0      4.0      0.0      BpG_M = math.ceil(M/32)
    32
    33                                               # Move training data to device and
        allocate arrays on device.
    34         1       4552.0   4552.0      0.1      d_Y = cuda.to_device(Y.reshape(-1)*1)
    35         1       6353.0   6353.0      0.1      d_vecproj = cuda.device_array(shape=(N*M*
        S))
    36         1       1898.0   1898.0      0.0      z = cuda.device_array(shape=(N*S))
    37         1        538.0    538.0      0.0      x = cuda.device_array(shape=(N*S))
    38         1        498.0    498.0      0.0      d_matproj = cuda.device_array(shape=(M*N)
        )
    39         1        499.0    499.0      0.0      d_signer = cuda.device_array(shape=(S*N))
    40
    41        21        101.0      4.8      0.0      for t in range(maxit):
    42        20       6550.0    327.5      0.1          N_timer.Timer(t, maxit)
    43      8020      33238.0      4.1      0.4          for n in range(N):
    44      8000     828387.0    103.5      9.6              I_D[:,n] = np.argpartition(np.abs
        (D.T@Y[:,n]), -S)[-S:]
    45        20       5464.0    273.2      0.1          D_new = np.zeros((M, K))
    46        20      43402.0   2170.1      0.5          DtD = D.T@D
    47
    48        20      49097.0   2454.8      0.6          d_D = cuda.to_device(D.reshape(-1)*1)
    49        20      71970.0   3598.5      0.8          d_DtD = cuda.to_device(DtD.reshape
        (-1)*1)
    50        20      30379.0   1519.0      0.4          d_I_D = cuda.to_device(I_D.reshape
        (-1)*1)
    51        20      28452.0   1422.6      0.3          d_Dnew = cuda.to_device(D_new.reshape
        (-1)*1)
    52
    53        20    4236604.0 211830.2     49.0          k_matvecproj[TpB, BpG_N](d_D, d_DtD,
        d_I_D, d_Y, d_vecproj, z, x, d_matproj, d_signer)
    54
    55        20    1018304.0  50915.2     11.8          k_updateD[TpB, BpG_M](d_Dnew, d_I_D,
        d_Y, d_vecproj, d_matproj, d_signer)
    56        20    1929974.0  96498.7     22.3          D_new = d_Dnew.copy_to_host()
    57        20        324.0     16.2      0.0          D_new = D_new.reshape(M, K)
    58
    59        20       6866.0    343.3      0.1          scale = np.sum(D_new*D_new, axis=0)
```

```
    60        20        835.0     41.8      0.0          iszero = np.where(scale < 0.00001)[0]
    61        20      12382.0    619.1      0.1          D_new[:,iszero] = np.random.randn(M,
        len(iszero))
    62
    63        20     302752.0  15137.6      3.5          D_new = normalize_mat_col(D_new)
    64        20       1591.0     79.5      0.0          D = 1*D_new
    65         1          3.0      3.0      0.0      return D
```

Listing K.3: Line profile of 2nd kernel implementation with nTrainingData set to 4000.

```
Timer unit: 2.77277e-07 s

Total time: 20.9684 s
File: GPU_itkrm_2.py
Function: gpu_itkrm at line 20

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    20                                           @profile
    21                                           def gpu_itkrm(data, K, S, maxit):
    22         1         15.0     15.0      0.0      M, N = data.shape
    23         1       5308.0   5308.0      0.0      D_init = np.random.randn(M, K)
    24       201        823.0      4.1      0.0      for i in range(K):
    25       200      19179.0     95.9      0.0          D_init[:,i] = D_init[:,i] / np.linalg
        .norm(D_init[:,i], 2)
    26         1          3.0      3.0      0.0      Y = data
    27         1         72.0     72.0      0.0      I_D = np.zeros((S, N), dtype=np.int32)
    28         1          4.0      4.0      0.0      D = D_init
    29         1          4.0      4.0      0.0      TpB = 32    # ThreadsPerBlock
    30         1         21.0     21.0      0.0      BpG_N = math.ceil(N/32)     #
        BlocksPerGrid
    31         1          6.0      6.0      0.0      BpG_M = math.ceil(M/32)
    32
    33                                               # Move training data to device and
        allocate arrays on device.
    34         1      67078.0  67078.0      0.1      d_Y = cuda.to_device(Y.reshape(-1)*1)
    35         1     137239.0 137239.0      0.2      d_vecproj = cuda.device_array(shape=(N*M*
        S))
    36         1       5491.0   5491.0      0.0      z = cuda.device_array(shape=(N*S))
    37         1       2195.0   2195.0      0.0      x = cuda.device_array(shape=(N*S))
    38         1       5971.0   5971.0      0.0      d_matproj = cuda.device_array(shape=(M*N)
        )
    39         1       2393.0   2393.0      0.0      d_signer = cuda.device_array(shape=(S*N))
    40
    41        21        103.0      4.9      0.0      for t in range(maxit):
    42        20       7305.0    365.2      0.0          N_timer.Timer(t, maxit)
    43    320020    1236899.0      3.9      1.6          for n in range(N):
    44    320000   30624444.0     95.7     40.5              I_D[:,n] = np.argpartition(np.abs
        (D.T@Y[:,n]), -S)[-S:]
    45        20       2845.0    142.2      0.0          D_new = np.zeros((M, K))
    46        20      42844.0   2142.2      0.1          DtD = D.T@D
    47
    48        20      57158.0   2857.9      0.1          d_D = cuda.to_device(D.reshape(-1)*1)
    49        20      37833.0   1891.7      0.1          d_DtD = cuda.to_device(DtD.reshape
        (-1)*1)
    50        20     196345.0   9817.2      0.3          d_I_D = cuda.to_device(I_D.reshape
```

```
    (-1)*1)
51        20      36043.0   1802.2      0.0          d_Dnew = cuda.to_device(D_new.reshape
    (-1)*1)
52
53        20    4270952.0 213547.6      5.6          k_matvecproj[TpB, BpG_N](d_D, d_DtD,
    d_I_D, d_Y, d_vecproj, z, x, d_matproj, d_signer)
54
55        20    1848808.0  92440.4      2.4          k_updateD[TpB, BpG_M](d_Dnew, d_I_D,
    d_Y, d_vecproj, d_matproj, d_signer)
56        20   36705224.0 1835261.2     48.5          D_new = d_Dnew.copy_to_host()
57        20        428.0     21.4      0.0          D_new = D_new.reshape(M, K)
58
59        20       7222.0    361.1      0.0          scale = np.sum(D_new*D_new, axis=0)
60        20        872.0     43.6      0.0          iszero = np.where(scale < 0.00001)[0]
61        20       1562.0     78.1      0.0          D_new[:,iszero] = np.random.randn(M,
    len(iszero))
62
63        20     298265.0  14913.2      0.4          D_new = normalize_mat_col(D_new)
64        20       1559.0     78.0      0.0          D = 1*D_new
65         1          4.0      4.0      0.0      return D
```

K.5.3 Execution Time

The execution time of the kernel implementation and the sequential implementation are shown in Table K.1 and Figure K.1.

Table K.1: Shows the execution times found through testing.

nTrainingData	2nd kernel impl. [s]	1st kernel impl. [s]	CPU [s]
100	2.40	4.54	3.89
1000	6.38	28.60	37.65
4000	20.97	110.89	149.91

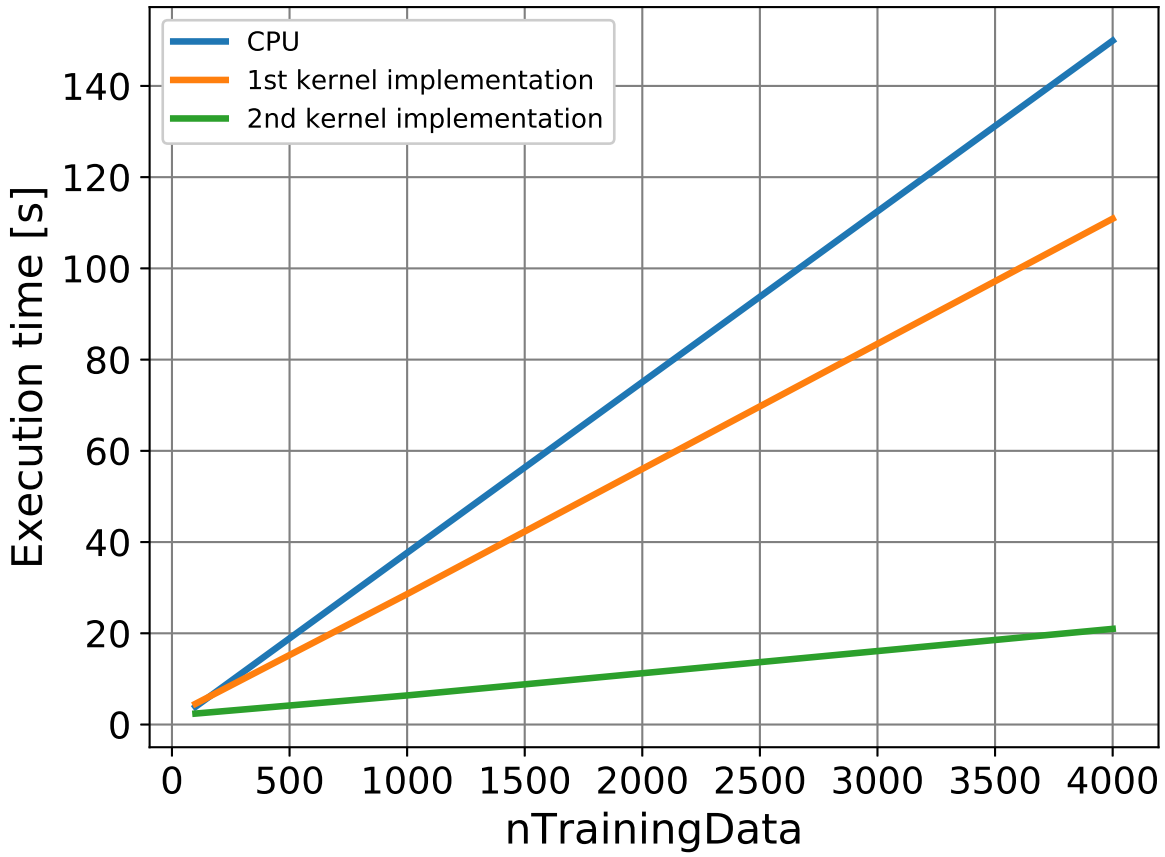


Figure K.1: Execution time of the kernel implementation compared with that of the sequentially optimized implementation.

K.5.4 Time used on data transfer

Listing K.4: nvprof of custom kernel for matrix and vector projections with nTraningData = 100

```
==1148== Profiling application: python gpu_itkrm_2.py
==1148== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 74.91%  420.20ms        20  21.010ms  19.752ms  24.011ms  cudapy::__main__::k_matvecproj$241(
     Array<double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>,
     Array<int, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>, Array<
     double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>, Array<
     double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>, Array<
     double, int=1, A, mutable, aligned>)
 24.62%  138.11ms        20  6.9053ms  6.4396ms  7.8754ms  cudapy::__main__::k_updateD$242(
     Array<double, int=1, A, mutable, aligned>, Array<int, int=1, A, mutable, aligned>, Array<
     double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>, Array<
     double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>)
  0.35%  1.9755ms        81  24.389us  5.2160us  63.709us  [CUDA memcpy HtoD]
  0.11%  629.73us        20  31.486us  31.230us  31.870us  [CUDA memcpy DtoH]
```

Listing K.5: nvprof of custom kernel for matrix and vector projections with nTraningData = 1000

```
1 ==12496== Profiling application: python gpu_itkrm_2.py
2 ==12496== Profiling result:
3 Time(%)      Time     Calls       Avg       Min       Max  Name
4  50.30%  1.30816s        20  65.408ms  64.349ms  74.600ms  cudapy::__main__::k_updateD$242(
      Array<double, int=1, A, mutable, aligned>, Array<int, int=1, A, mutable, aligned>, Array<
      double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>, Array<
      double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>)
5  44.99%  1.17014s        20  58.507ms  57.994ms  59.123ms  cudapy::__main__::k_matvecproj$241(
      Array<double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>,
      Array<int, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>, Array<
      double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>, Array<
      double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>, Array<
      double, int=1, A, mutable, aligned>)
6   4.68%  121.83ms        81  1.5041ms  25.311us  119.04ms  [CUDA memcpy HtoD]
7   0.02%  632.23us        20  31.611us  31.359us  34.174us  [CUDA memcpy DtoH]
```

Listing K.6: nvprof of custom kernel for matrix and vector projections with nTraningData = 4000

```
1 ==12936== Profiling application: python gpu_itkrm_2.py
2 ==12936== Profiling result:
3 Time(%)      Time     Calls       Avg       Min       Max  Name
4  48.28%  5.17227s        20  258.61ms  257.85ms  265.64ms  cudapy::__main__::k_updateD$242(
      Array<double, int=1, A, mutable, aligned>, Array<int, int=1, A, mutable, aligned>, Array<
      double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>, Array<
      double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>)
5  47.33%  5.07087s        20  253.54ms  248.19ms  255.14ms  cudapy::__main__::k_matvecproj$241(
      Array<double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>,
      Array<int, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>, Array<
      double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>, Array<
      double, int=1, A, mutable, aligned>, Array<double, int=1, A, mutable, aligned>, Array<
      double, int=1, A, mutable, aligned>)
6   4.39%  470.07ms        81  5.8033ms  25.343us  454.20ms  [CUDA memcpy HtoD]
7   0.01%  629.12us        20  31.456us  31.231us  31.647us  [CUDA memcpy DtoH]
```

L | Visual Performance

Name: Group 871
Date: 02/05 - 2018

L.1 Purpose

A visual performance test is made of the ITKrM algorithm. The visual performance measure is the SSIM index and this is set in relation to the size of the sub images for training the algorithm.

L.2 Computer Specifications

Platform	Type
Computer	Lenovo Thinkpad E470
CPU	Intel Core i7-7500U Processor (2.70GHz 4MB)
GPU	NVIDIA® GeForce® 940MX 2 GB
RAM	8.0GB DDR4 SODIMM 2133MHz

L.3 Preliminary

In the following test multiple files are used. These files contains code for the ITKrM algorithm, a main file to run the test and a file containing the training data (image data). A file containing the OMP algorithm is also needed. All of these files can be found in `project_code\L`, and they are named:

- Main: combined_test_MRI.py
- ITKrM: ITKrM_parallel_map.py
- Training data: mri_scans_shaped.npy
- OMP: OMP_fast.py

In the file called combined_test_MRI.py different variables needs to be initialized:

Variable	Value
K	200
S	[80, 40, 20, 5, 1]
maxit	20
N	16384
nTrainingData	50
data	data = LoadFromDataBatch.ImportData(-1, 1) data = data[:nTrainingData, :]
W_data	128
H_data	128
N_subpic	[64, 32, 16, 8]

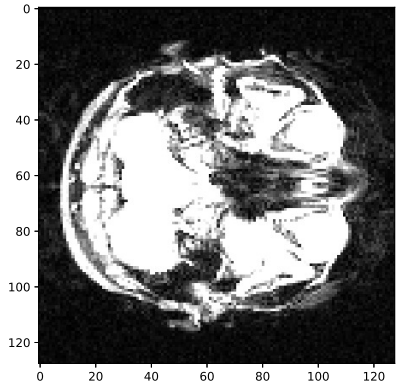
L.4 Procedure

The test is made using the following steps:

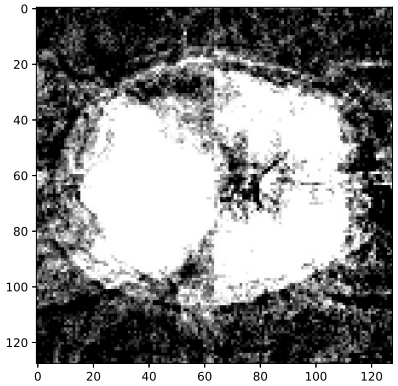
1. Set N_subpic = 64 and S = 80 and run the file combined_test_MRI.py.
2. Save the beforeImage and the afterImage.
3. Repeat step 1 and step 2 with N_subpic = $[32, 16, 8]$ S = $[20, 5, 1]$
4. Repeat step 1, 2 and 3, but with S held constant to 40.

L.5 Results

Visual performance test with relative S.
The results of the visual performance test with relative S is presented in Figure L.1, Figure L.2, Figure L.3 and Figure L.4.

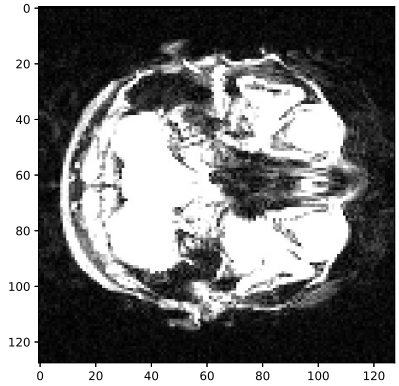


(a) beforeImage.

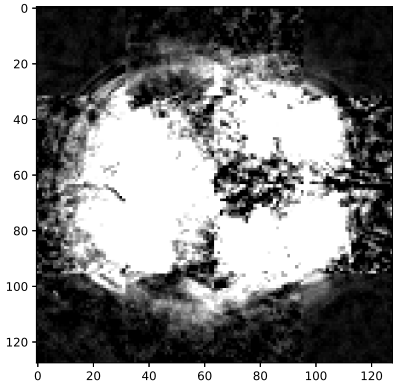


(b) afterImage with N_subpic = 64 and S = 80.

Figure L.1: Results visual performance test with N_subpic = 64 and S = 80. SSIM index = 0.365.

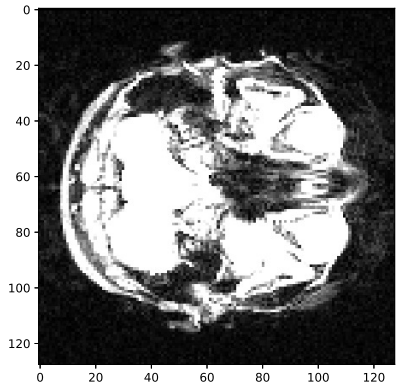


(a) beforeImage.

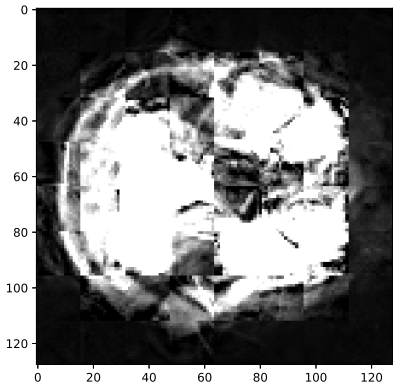


(b) afterImage with N_subpic = 32 and S = 20.

Figure L.2: Results visual performance test with N_subpic = 32 and S = 20. SSIM index = 0.435.

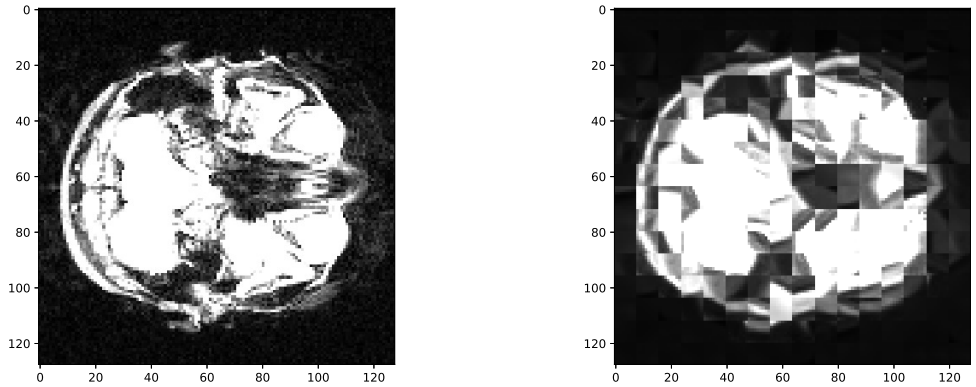


(a) beforeImage.



(b) afterImage with N_subpic = 16 and S = 5.

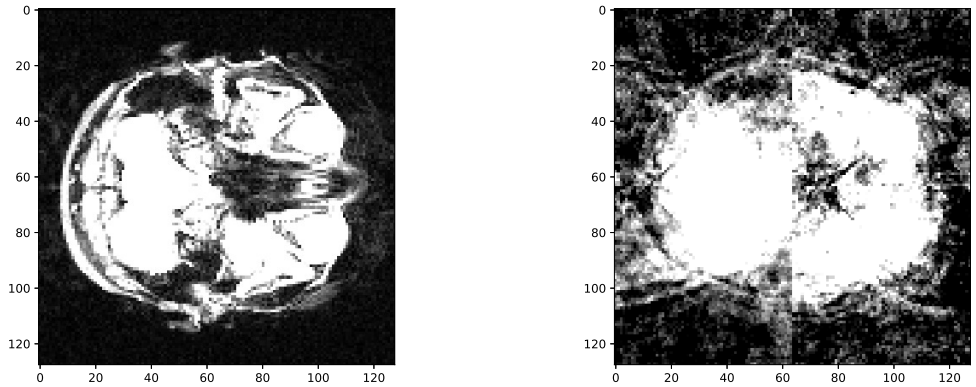
Figure L.3: Results visual performance test with N_subpic = 16 and S = 5. SSIM index = 0.567.



(a) beforeImage. **(b)** afterImage with N_subpic = 8 and S = 1.

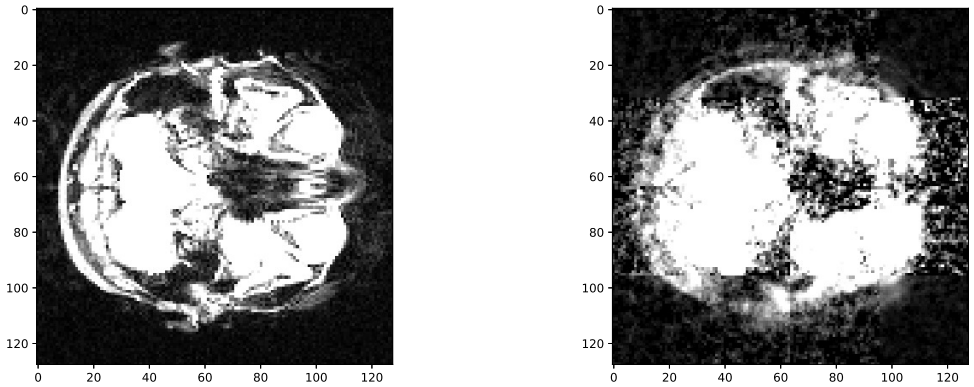
Figure L.4: Results visual performance test with N_subpic = 8 and S = 1. SSIM index = 0.603.

Visual performance test with constant S.
The results of the visual performance test with constant S is presented in Figure L.5, Figure L.6, Figure L.7 and Figure L.8.



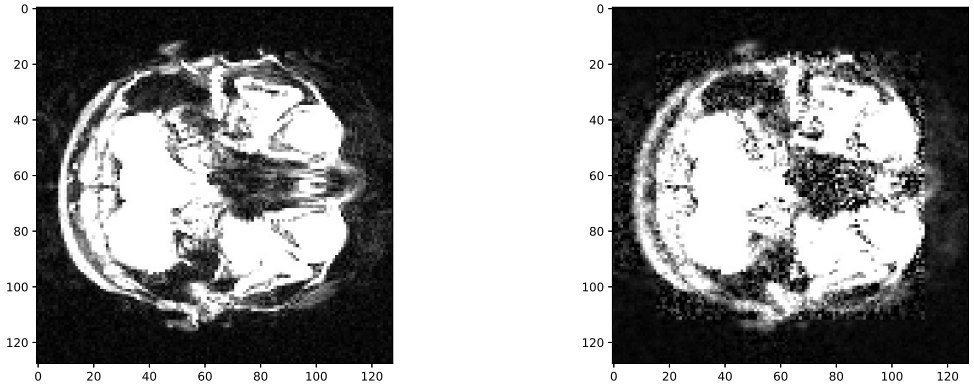
(a) beforeImage. **(b)** afterImage with N_subpic = 64 and S = 40.

Figure L.5: Results visual performance test with N_subpic = 64 and S = 40. SSIM index = 0.327.



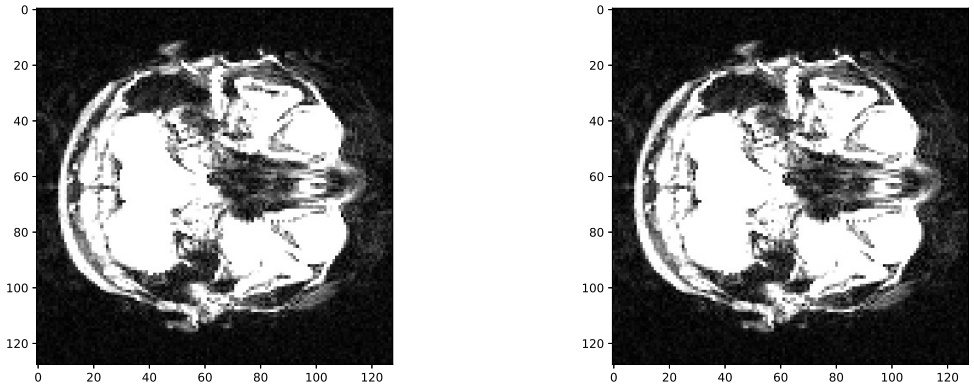
(a) beforeImage. **(b)** afterImage with N_subpic = 32 and S = 40.

Figure L.6: Results visual performance test with N_subpic = 32 and S = 40. SSIM index = 0.469.



(a) beforeImage. **(b)** afterImage with N_subpic = 16 and S = 40.

Figure L.7: Results visual performance test with N_subpic = 16 and S = 40. SSIM index = 0.720.



(a) beforeImage. **(b)** afterImage with N_subpic = 8 and S = 40.

Figure L.8: Results visual performance test with N_subpic = 8 and S = 40. SSIM index = 0.994.