Optimisation of a hippocampus model with test-driven GPU parallel programming

UNIVERSITY OF
LINCOLN

Jack Stevenson

School of Computer Science

College of Science

University of Lincoln

Submitted in partial satisfaction of the requirements for the
Degree of Master of Science Intelligent Vision
in Computer Science

Supervisor Dr. Charles Fox

August 2020

Abstract

The Hippocampus is a useful biological model of spatial memory. Functioning models of hippocampus may be useful for robot navigation. Previous work, (Fox and Prescott, 2010a; Fox and Prescott, 2010b; Saul, Prescott and Fox, 2011) developed a computer model to mimic the hippocampus, implemented in serial code. However, serial execution is very slow at scale. This means that the model only functioned in very small 'toy' environments. Modern GPUs can massively parallelize execution of neural networks. This is usually done for the backpropagation of multilayer perceptrons, but they similarly offer the possibility of speeding up more realistic biological networks such a this hippocampus model. This project aims to modify the model's existing serial learning process to be more parallel, using GPU hardware. Our research finds that the model may be suited to using the CPU for small environments, however in simulated larger environments that the model may need to learn would greatly benefit from parallel execution.

Acknowledgements

I would like to thank my supervisor, Charles Fox for helping to steer me away from my perceived biases and pushing me to not just settle on adequate results.

Table of Contents

1 Introduction **1**

2 Background and Related Work **2**

2.1 Review of Hippocampus models 2

2.1.1 Hippocampus biology 2

2.1.2 Hippocampal model - Unitary Coherent Particle Filter 2

2.2 Parallel Computing Technologies 4

2.2.1 Low level 4

2.2.2 High level parallel libraries for Python 7

2.3 Software Engineering and refactoring 10

2.3.1 CPU profiling 11

2.3.2 GPU profiling 13

2.3.3 Unit tests 15

2.3.4 Refactoring with Unit Tests 16

3 Methods, Experiments and results **18**

3.1 Process of refactoring 19

3.2 Evaluating the refactoring process 20

3.3 Unit Test Results 21

3.4 Function Timing 23

3.5 Artificially expanding the size of the CA3 sub-field 24

3.5.1 Results from increasing the size of the CA3 sub-field 25

3.6 Optimisation of data type casting 26

3.6.1 Results from optimising the casting operation 26

3.7 Optimisation of the error function 27

3.7.1 Results from optimising the error function 27

3.8 Profiling Results 28

3.8.1 Number of calls 28

3.8.2 Total time 29

3.9 Verification of model accuracy 31

4 Discussions and Conclusions **33**

References **35**

A Additional assessment information **38**

A.1 Aims and objectives 38

A.2 Tools and Toolsets 38

A.2.1 Development Environment 39

A.2.2 Git and Github 39

A.2.3 Latex and Overleaf 39

A.3 Project Management 40

A.4 Reflection 41

List of Figures

2.1 Visual representation of the hippocampal circuit, modified from the Cajal, 1911 original by Looie496, 2008. 3
2.2 Illustration of the hippocampus model showing data flows and hippocampus regions. SURF features are the new visual inputs. (Subiculum circuit not shown.) from Saul, Prescott and Fox, 2011 3
2.3 Sample output from the Python Profile module 12

3.1 Unit test results for the NumPy version on CPU of the `boltzmannProbs` function. 21
3.2 Unit test results for the Tensorflow version on GPU of the `boltzmannProbs` function. 22
3.3 The plus maze environment the model is learning, from Fox and Prescott, 2010a 24
3.4 Number of calls for the CPU implementation of the model. 28
3.5 Number of calls for the GPU implementation of the model. 28
3.6 Total time for function calls for the CPU implementation of the model. 29
3.7 Total time for function calls for the GPU implementation of the model. 30

List of Tables

3.1 Table of results comparing different functions as we parallelise the learning process. 23

3.2 Time taken to run the training process for different sizes of an artificially inflated CA3 using the model variations from test 0 and test 17. 25

3.3 Timing 100000 invocations of each function over different numbers of nodes. The times reported are cumulative of all 100000 invocations. 26

3.4 Timing 100 invocations of the named functions across different numbers of nodes. The times reported are cumulative of the 100 invocations. 27

3.5 Table comparing the accuracy of the model with respect to a 3000 step walk in the plus maze for each type of learning method, using handset weights, learned weights and random weights. 31

A.1 Risk matrix for project. 41

Chapter 1: Introduction

The hippocampus is an important area of the brain involved in spatial memory. This allows us to consider it as a method of robot navigation. This can be done by mapping the specialised cells in the hippocampus to activations in a machine learning system. However this approach can be considered slow due to the serial learning process, and is unsuitable for real-time use. Despite this, there are sections of the model which would benefit from parallelisation, making a real-time system using this approach more viable.

One model of the hippocampus is the unitary coherent particle filter, as proposed by Fox and Prescott, 2010a. This model utilises the wake-sleep algorithm proposed by Hinton, Osindero and Teh, 2006. This is useful as it allows for the model to respond to immediate changes to inputs. The model architecture is explored in more detail in Section 2.1.2 .

We can use modern compute technology, such as GPUs to parallelise the learning process of the model. By parallelising our code, we enable the easy horizontal scaling of the model by reducing the workload on a given processing unit. As part of this process, we will need to re-implement the code to allow for a parallel implementation of the serial learning process.

A process for re-implementing code is refactoring. This involves re-writing code without changing the external behaviour of the code. This is beneficial to our project as it provides us a methodology to maintain behaviour when as we switch the model to use a parallel implementation.

In this thesis, we will refactor the Hippocampal model of Saul, Prescott and Fox, 2011 into a new parallel TensorFlow based implementation. We do this by wrapping the functions in unit tests, which will enable us to make the model run faster and be used on robots in real time.

Chapter 2: Background and Related Work

2.1 Review of Hippocampus models

2.1.1 Hippocampus biology

The hippocampus Andersen et al., 2006 is involved in the creation and consolidation of memories, in a circuit along with the Entorhinal Cortex (EC) and Dentate Gyrus (DG). The main function of the hippocampus is thought to be the consolidation of associative memories and spatial memory. As such it can be considered suitable for navigation within an environment, associating each place with sensory memories about that place.

The hippocampus is comprised of four main sub-fields: CA1, CA2, CA3 and CA4. However, the classical view of the hippocampal circuit only considers the CA1 and CA3 sub-fields, along with the EC and DG. The circuit accepts input from external sources and the response from the CA1 sub-field. This is then passed to the DG which sparsifies the EC output. The output of the EC is also passed to the CA3 sub-field along with the output of the DG. The CA3 sub-field also accepts recurrent connections. This allows it to consolidate interactions between old and new stimuli. CA3 then projects its output to the CA1 sub-field, which provides the main output to the rest of the limbic system. This structure is shown in Figure 2.1

2.1.2 Hippocampal model - Unitary Coherent Particle Filter

The initial model of the unitary coherent particle filter of hippocampus (UCPF-HC) by Fox and Prescott, 2010a uses handset weights to initialise the model and understand how it initially evolves. This is further expanded to use sensors to differentiate between locations within a simple environment.

The model is comprised of 4 main areas, an EC representation, a DG representation, the CA3 subfield and the CA1 subfield. The EC and DG are used to provide input

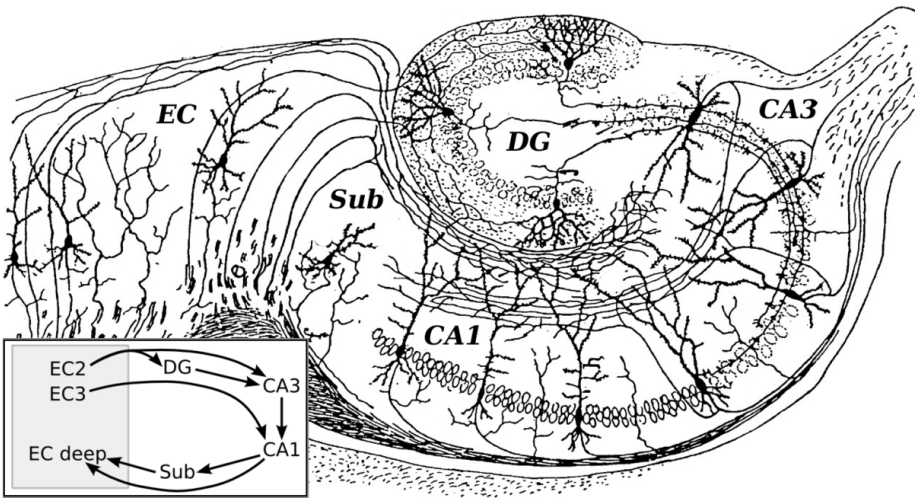

Figure 2.1: Visual representation of the hippocampal circuit, modified from the Cajal, 1911 original by Looie496, 2008.

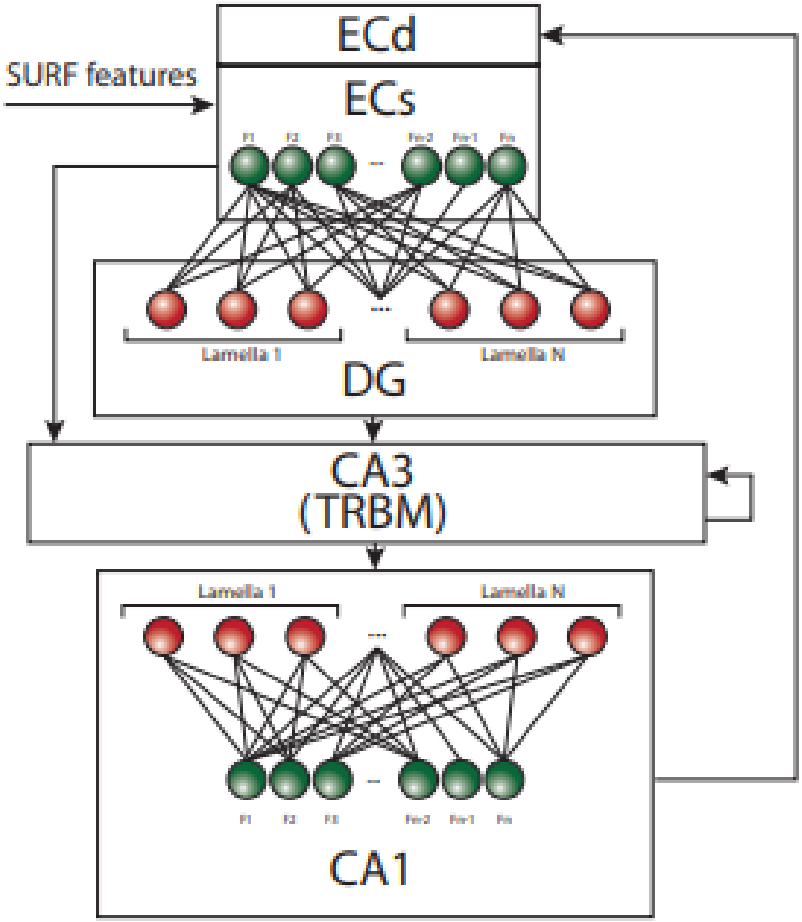

Figure 2.2: Illustration of the hippocampus model showing data flows and hippocampus regions. SURF features are the new visual inputs. (Subiculum circuit not shown.) from Saul, Prescott and Fox, 2011

from the outside world to the model, encoding them into lamellae. They project to the CA3 sub-field, which is represented in the model by a modified Temporal Restricted Boltzmann Machine(TRBM), proposed by (Hinton, Osindero and Teh, 2006). This allows the CA3 sub-field to be represented by the joint probability distribution:

$$P(x_t, x_{t-1}, z_t) = \frac{1}{Z} \exp \sum_t (-x'_t W_{x'_t x'_t} x'_{t-1} - x'_t W_{x'_t z'_t} z'_t) \tag{2.1}$$

In this distribution, z represents a boolean observation vector, x represents a boolean hidden state vector and the weight matrices between them. This distribution, in conjunction with the modification of the standard Temporal Restricted Boltzmann Machine allows for a deterministic update, obtaining maximum *a posteriori* estimates,

$$\hat{x_t} \leftarrow arg\max P(x_t | \hat{x_{t-1}}, z_t) = \{\hat{x_t}(i) = (P(x_t(i) | \hat{x_{t-1}}, z_t) > \frac{1}{2})\}_i \tag{2.2}$$

which is considered to be a zero temperature limit of the annealed sequential Gibbs sampler. This sampler gives a set of boolean values, which can then be used to identify where the model believes it is in the environment. These boolean values can then be decoded into the CA1 sub-field, which projects back to the EC, completing the circuit.

2.2 Parallel Computing Technologies

We have currently reviewed the biology of the hippocampus and a model that uses it. This model currently uses a serial process to learn. The aim of the study is to make the model learn faster in computational implementations. One method of making them go faster is parallelisation. However there are many forms of parallelisation technology. As such we will need to choose one, focusing on GPUs only. So this section will review these options and make a choice on the technology to use.

2.2.1 Low level

CUDA

CUDA is a proprietary parallel framework for interfacing with NVidia GPUs. It is a C-based programming language, and uses kernel functions to run code in parallel. CUDA handles the launching of kernels and memory management through the use of the CUDA runtime(Nvidia C.U.D.A., 2020). The CUDA runtime makes the assumption that memory on the host (CPU/RAM) and memory on the device (GPU) are separate. This means that data needs to be transferred between the CPU and

GPU when the GPU is needed. This creates overhead from the copy operations, as well as spawning the CUDA runtime process. On the other hand, this approach to the initialisation of CUDA means that GPU is only in use when needed, and can be considered free when not in use, allowing multiple programs access to the same hardware.

OpenCL

OpenCL is an open-source parallel framework for interfacing with GPUs. As an open framework, it is capable of working on both NVidia GPUs and AMD GPUs. However there are only a few functions in the standard. This means that external libraries, such as clBLAS (clMathLibraries, 2020) or CLBlast (Nugteren, 2017) are needed to aid machine learning applications.

```
//Initialise OpenCL, run once
context -> CL_Context
queue -> CL_Command_Queue(context)
kernel_source -> CL_Program_Source(readfile(kernels))
program -> CL_Program.of(kernel_source)
program.build()

//Run a kernel
inputA -> vector<int>[N]
inputB -> vector<int>[N]
output -> vector<int>[N]
bufferA -> CL_Buffer.ofSize(length(inputA)*sizeof(int))
bufferB -> CL_Buffer.ofSize(length(inputB)*sizeof(int))
bufferC -> CL_Buffer.ofSize(length(output)*sizeof(int))
queue.writeVectorToBuffer(inputA, bufferA)
queue.writeVectorToBuffer(inputB, bufferB)
kernel -> CL_Kernel(program, KERNEL_FUNCTION_NAME)
setKernelArgument(kernel, bufferA)
setKernelArgument(kernel, bufferB)
setKernelArgument(kernel, bufferC)
queue.runKernel(kernel)
queue.readBufferToVector(bufferC, output)

return output
```

Code Segment 2.1: Psuedocode of CL Initialisation and Kernel Execution

When programming for pure OpenCL, a developer is often writing the code that directly interfaces with the GPU. The libraries reduce the amount of code that needs writing by providing optimised routines, however these routines often perform a single function. Thus they are not suitable for neural network calculations as

each operation would need to be transferred between the CPU and GPU, increasing overhead.

In addition, the OpenCL runtime requires a large amount of setup code to begin working with it. The setup code is trivial to create, as seen in Code Segment 2.1. However it adds a large amount of preparation code for each call to a parallel kernel. This increases the amount of code for a given function, which is likely to increase overall running time, and reduce the readability of the function.

As OpenCL is able to run on multiple brands of GPUs, there are many different implementations of the standard. However, as CUDA is directly tied to NVidia GPUs, there is only a small distinction between the CUDA API and CUDA Runtime, as it is closely tied to NVidia GPUs. There have been attempts to integrate the CUDA API onto other devices, using OpenCL as the base for it's processing. We discuss a couple of these implementations below.

ROCm/HIP

ROCm is a low level parallelisation framework for AMD Hardware, developed by AMD. It uses a single source model. This allows it to work on different hardware platforms, such as CPUs, APUs and GPUs.

It has different variations depending on the target platform and applications, such as servers (ROCk) and machine learning (mlOpen). This allows it to be similar to CUDA. It also supports HIP code, which allows ROCm programs to run on CUDA-enabled GPUs.

However, it is a relatively new technology, in comparison to CUDA and OpenCL insights and strategy, 2016. This means that it does not have the same amount of support as CUDA for machine learning. This makes it unsuitable for our project, as it can require a large amount of setup compared to using native CUDA hardware.

SYCL

In contrast to OpenCL and CUDA, SYCL is a relatively new low level platform for parallel computing. It builds upon the OpenCL standard, however unlike OpenCL it incorporates high level routines as well. As such, it sits between the code and other low-level architectures such as OpenCL, CUDA and OpenMP.

SYCL programmes use a single source paradigm. This allows for parallel kernels to be included as part of the original source. This reduces the overhead for loading the kernels into the program, unlike OpenCL. It also means that code written for SYCL can be compiled for different platforms, such as CUDA and HIP/ROCm. By allowing the compiler to handle how the code is compiled and initialised, it can be assumed that the routines used are heavily optimised for the architecture.

Like ROCm, SYCL is relatively new. As such, there is not as much support for SYCL implementations of higher level libraries. There has been an attempt to port Tensorflow, which originally uses a CUDA backend, to a SYCL b backend (find sycl tensorflow), however it is in the alpha stages, thus it may not be stable enough for our purposes.

2.2.2 High level parallel libraries for Python

There are different parallel libraries for high level languages which can be used to bind native language code, such as Java and Python, to the low level libraries like CUDA and OpenCL. As the computational implementation of the UCPF model by Saul, Prescott and Fox, 2011 uses Python, we consider a subset of Python bindings as there are many different bindings in various stages of implementation compliance.

PyOpenCL

PyOpenCLKlöckner et al., 2012 is a binding to the OpenCL library. Kernels are written as strings, and the PyOpenCL library compiles them into executable code. This provides the benefit of being able to easily write and understand the kernels. However, software written with OpenCL requires a lot of setup in order to execute it, as seen in Code Segment 2.1.

PyOpenCl only provides bindings to the OpenCL API, thus external libraries such as clBLAS would need to be imported as well to make use of them, which can be considered a non-trivial task. In addition, many machine learning backends, e.g. Keras and Theano, focus more on CUDA based implementations, as opposed to an OpenCL implementation. Implementing functions of the UCPF-HC model as OpenCL kernels would optimise the model, however it would also increase the overall complexity of the software, an outcome that refactoring aims to avoid. This makes PyOpenCL unsuitable for our project.

CuPy

CuPy is an array manipulation library accelerated by CUDA. It aims to be a NumPy compatible library (Okuta et al., 2017). This means it provides functions that operate the same as NumPy functions. This is useful as it would make the transition from NumPy, which is a serial array manipulation library that the model currently uses, to a parallel system easier.

However part of our development process is to create a system that is easy to update in the future. Whilst CuPy could be used as a parallelisation framework, it's similarity to the current serial implementation, NumPy, does not push the boundary far enough to allow for different frameworks to be experimented with. Furthermore,

CuPy does not integrate with machine learning backends, such as Keras. This is detrimental because it does not allow for future builds of the backends, which include Restricted Boltzmann Machines and other machine learning architectures aside from neural networks to be easily integrated with out existing model, provided the implementations are able to pass the tests. In addition, CuPy does not scale across multiple GPU devices, which indicates that versions of the model with several thousands of neurons would be incompatible with using CuPy as the backend for handling the learning.

Theano

Historically, the first widely used data flow graph system was Theano. Theano evolved over several years and eventually influenced newer libraries such as TensorFlow and Torch. Both of these libraries have replaced Theano but they still owe many architectural and implementation features to it.

Theano works by defining mathematical operations and data points as nodes on a directed acyclic graph (Al-Rfou et al., 2016). This graph is then compiled to a highly optimised form. By compiling the graph, it allows for reduced memory consumption, and the transfer of the model to a hardware accelerated device, such as GPUs. GPU computation is done on CUDA enabled devices.

This parallel computation makes Theano suitable for vector and matrix calculations, which allows it to be useful for neural network calculations. This makes Theano suitable for our purposes as our model heavily uses matrix calculations as part of it's calculations.

However, Theano is an older library, with support from the official maintainers, MILA stopped as of 2017 (Bengio, 2017). This means that it is less likely to support current hardware, and will not support future hardware. As such, we will not be using Theano for this project as our intent is to allow for future parallelisation of the model.

PyTorch

PyTorch is a set of Python bindings to the Torch Machine Learning Tensor library, developed by Facebook. As a tensor library, it has mathematical functions implemented, as well as machine learning algorithms. It is primarily developed for batch processing, using custom cache allocations to reduce synchronisation barriers and maximise device usage.

A large proportion of the PyTorch machine learning functionality is provided by the libTorch library. This allows for efficient operations as libTorch is written in C/C++. This does cause some overhead compared to working in pure Python, however the

highly optimised C code negates the overhead. In addition the Python language does have the ability to compile it's code as C code. This makes the overall processing much faster, and more suitable for deployment purposes.

The authors of PyTorch focus on the philosophy of 'everything is a program' Paszke et al., 2019. This suggests that software written for PyTorch can be considered as a collection of small, independent programs. By decoupling implementation from usage, the philosophy makes it suitable for unit testing as each program can be tested independently. In addition, testing can be performed on the integration of these programs.

Applying the philosophy of PyTorch to our system, we have three major programs: the creation of the environment, the learning process and the verification process, which would simplify our system for future work, as it can be targeted for a particular subsystem, with full integration being tested near the end of the process. However, implementing the model in this manner greatly increases the scope of our refactoring, and does not benefit our optimisation focus.

Tensorflow

Like Theano, Tensorflow is a tensor based machine learning library that is being developed by Google. It originally used a graph-based execution model, with tensor operations being defined as nodes on the graph. However as of Tensorflow 2 it uses an eager execution model. This makes it useful for prototyping, however eager execution also reduces the overall optimisation of the model as it is interpreting the human-readable code as opposed to the machine readable binary.

```
input_1 = tf.Variable(shape=(5,1))
input_2 = tf.Variable(shape=(1,3))
operation_matmul = tf.matmul(input_1, input_2)
operation_sum = tf.reduce_sum(operation_matmul)
```

Code Segment 2.2: Tensorflow Code Example

As Tensorflow 2 is a machine learning library, it primarily works with the Keras backend. This is because Keras is a simple to use framework for learning deep netwroks. Due to the versitility of deep learning in performing different tasks, such as Writing Music (Dhariwal et al., 2020), Voice Puppetry (Thies et al., 2020) or Medical Analysis (Wang et al., 2020), it makes deep models extremely useful outside of research based tasks. In contrast to deep learning, our model is considered shallow. This means that we will not be working with Keras or deep learning in our study, however having them as an option for future work is beneficial. This is because the developers of these libraries may implement functions for Restricted Boltzmann Machine Learning in the future, and these will likely be highly optimised by being low level functions with high level invokers.

Despite the focus on Deep learning and neural networks with Keras (Tensorflow, 2020), Tensorflow can also be used as a maths library. This is useful for our purposes as it can act as a replacement for NumPy, without being extremely similar. It also allows the model to operate on a different maths framework which should enable easier porting of the network to different frameworks/libraries.

Choice of framework

A study by (Bahrampour et al., 2015) compares different machine learning frameworks. The study looks at five frameworks, Caffe, Neon, Theano, Torch and Tensorflow. Their results indicate that in terms of speed, Theano and Torch are the best options, for small and large networks respectively. However this study was published in 2016, which means that the results may not reflect current standards in these libraries, such as Theano discontinuing development and the release of Tensorflow 2.0.

The study also indicates that Tensorflow is shown to have the most flexibility. This flexibility makes it simpler to create software with it as it allows for variances when refactoring the model. It should also be noted that GPU tests in the study use Tensorflow with cuDNN v2, whereas the other listed libraries use cuDNN v3. This was because these were the officially supported versions, thus optimisations in cuDNN versions may have affected the results.

Another reason for choosing Tensorflow over other machine learning libraries is that it allows for low level data flow graphs to be generated, whereas many other machine learning libraries for Python focus on Multi-layer Perceptron model architectures. It is more beneficial to have control over the data flow graphs for this model as we are using modified versions of the standard Temporal Restricted Boltzmann Machine equations.

2.3 Software Engineering and refactoring

Our study is aiming to speed up the computational implementation of a hippocampus model. But the main problem with speeding up software is knowing what should be targeted for speeding up. The slower section of the code are known as bottlenecks, because they limit the number of operations that can be performed. Amdahl's law describes the relationship between the parallelisable section and serial section of code.

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{2.3}$$

The formalisation of Amdahl's law (as seen in (2.3)), shows that the theoretical speed up is proportional to the amount of code that can be parallelised. Therefore, it is important to find as many bottlenecks as possible, and speed them up one at a time.

To enable this, profilers are tools which monitor the process executing the program, and time individual calls to functions. This is useful as it provides a numerical value to each function, including library calls, which can be used to identify the bottlenecks in the code. In this section we compare different profilers, and select an profiler to use.

2.3.1 CPU profiling

There are two modules for profiling in Python, `Profile` and `cProfile`. The `Profile` module is used for pure Python code, where as the `cProfile` module is used for C-based Python code. Both modules share the same interface and output format, thus the decision of using one over the other is dependent on speed.

As C is considered the most optimised language, moving a lot of our code into the C layer, which is done by the high level parallel libraries above, is important. This makes the profiler of choice `cProfile`, which can record and report functions that work in either the C layer or the Python layer.

The profiler provides six columns of information for the program, seen in Figure 2.3. These are:

- Number of calls to the function (nCalls), which indicate the number of times a function is invoked,
- Total time spent in the function, excluding calls to other functions,
- Cumulative time spent in the function, including calls to other functions,
- Function name and location, which help identify where the profiled function in the code is.
- Two columns of the average time taken per call, for each of the types of timing.

At this level, we can disregard the cumulative time and average time columns as they do not definitively indicate slow areas of code. This is because the slow parts of cumulative time can be attributed to a different function. As for average times, by reducing the total time for the function to execute the average time will also fall. The nCalls column is of less importance than the total time column as it does not actively contribute to identifying slow code, but can point to where our time should be focused in terms of optimisation.

As our goal is to reduce the global time for the system to learn, the total time column can be considered the most important measure of identifying optimisation

```
         176102019 function calls (174936111 primitive calls) in 813.799 seconds

   Random listing order was used
   List reduced from 9696 to 91 due to restriction <'cffun|DGStateAlan|go.py|gui|hcq|learnWeights|
location.py|makeMake|paths|plotPlaceCells|rbm|SURFExtractor'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 plotPlaceCells.py:1(<module>)
    30000    0.991    0.000   23.846    0.001 learnWeights.py:13(err)
        1    0.000    0.000    0.000    0.000 learnWeights.py:26(<listcomp>)
        1    0.000    0.000    0.000    0.000 learnWeights.py:37(<listcomp>)
        1    9.132    9.132  779.271  779.271 learnWeights.py:21(learn)
        1    0.000    0.000    0.001    0.001 learnWeights.py:1(<module>)
        1    0.000    0.000    0.001    0.001 rbm.py:25(trainPriorBias)
        3    1.231    0.410   80.129   26.710 rbm.py:34(trainW)
       90    1.408    0.016    1.477    0.016 rbm.py:87(hardThreshold)
        4    0.000    0.000    0.008    0.002 rbm.py:94(addBias)
        1    0.000    0.000    0.001    0.001 rbm.py:1(<module>)
        1    0.000    0.000    0.006    0.006 DGStateAlan.py:17(__init__)
```

Figure 2.3: Sample output from the Python Profile module

possibilities. This is because it ignores other factors and only reports the time taken, with respect to some overhead from the profiler tool. By using the total time and nCalls metrics, it allows for more time to be spent on doing the refactoring and testing, as opposed to guessing where to optimise.

Line Profiler

Line profiler is a Python Library for profiling code on a line by line basis, instead of the entire software. This allows for targeted profiling of the code. The targeting of code profiling makes it useful for finding bottlenecks in identified functions.

The Line profiler tool provides a percentage of the total time each line uses in the function. This percentage is useful for identifying the bottlenecks, as a larger percentage indicates that more time is needed for a given line.

An example of line by line profiling can be seen in Code Segment 2.3. Lines 10 through 14 only take a few microseconds, which has a negligible effect on the total percentage of time required. However further down the function, lines 17-19, take approximately 18 seconds each, which is around 3.5% of the total time, which points to optimising the `trainW` function. By optimising the `trainW` function, we reduce the overall runtime of the `learn` function, which aids our goal of optimising the learning process of the model.

The Line Profiler module is not part of the standard Python library. It is only being maintained to current Python versions. Thus it is not actively improved. This may mean that it is not as accurate as the profiling modules. Despite this, using it for identification purposes for large discrepancies in time for each line of the function.

```
Scale: 1e-6s
Total time: 529.294 s
File: /home/yakkus/github/jack-fork/hclearn/workspace/src/main/
   learnWeights.py
Function: learn at line 21

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    21                                           @profile
    22                                           def learn(path,
   dictSenses, dictGrids, N_mazeSize, ecs_gnd, dgs_gnd, ca3s_gnd,
   b_learnIdeal=True, b_learnTrained=False, b_learnDGWeights=True,
   learningRate=0.01):
    23         1          4.0      4.0      0.0      dghelper=None
    24  #Learn DG weights when you have visual input
    25         1          3.0      3.0      0.0      if
   b_learnDGWeights:
    26  #Extract data from dictionary of senses to train on
    27         1         14.0     14.0      0.0          allSURFS =
   [ sense.surfs for sense in
    ...
    59
    60         1   19120224.0 19120224.0      3.6          WR =
   trainW(lag(hids,1), hids, WB, N_epochs=1, alpha=0.01)
    61         1   18793573.0 18793573.0      3.6          WS =
   trainW(senses,      hids, WB, N_epochs=1, alpha=0.01)
    62         1   17906180.0 17906180.0      3.4          WO =
   trainW(odom,        hids, WB, N_epochs=1, alpha=0.01)
```

Code Segment 2.3: Example output from the line profiler tool, showing part of the learn function of the model.

2.3.2 GPU profiling

As with the low level parallelisation libraries, there is no set standard of profiling on a GPU, which is restricted by how it communicates with the CPU. This means that profilers for a given implementation, such as CUDA or OpenCL, only work for that implementation. We look at the methods of profiling on different GPU architectures, as a reference for the benefits and drawbacks for each architecture for future work to consider.

Profiling CUDA based software

The CUDA parallel runtime uses two main software packages to handle parallel profiling: NSight Compute and NSight Systems.

NSight Compute profiles each CUDA kernel run, similar to the CPU profiling methods, which help identify which operations are the bottlenecks. It also shows the dependencies of the kernels, such as a matrix multiplication kernel for input matrices of shape (m, k) and (k, n) using a vector multiplication kernel for each data point in the output matrix of shape (m, n). It also shows how memory is used across the GPU, as it is faster to access local memory to a processing unit instead of global GPU memory. This is useful as CUDA kernels are C-based, meaning that it can help detect a memory leaks, however, in our case the CUDA kernels are managed via the Tensorflow library. This particular feature is not useful to us at this time, however if further work is to directly use CUDA, then it would be necessary to ensure that the horizontal scaling of the model does not cause out of memory errors.

In contrast, NSight Systems looks at algorithm performance. This can be considered the utilisation amount of the GPU and the amount of time waiting for threads to synchronise. By understanding how the data is distributed across the GPU, it helps us understand whether a given function would benefit from being parallelised. Parallel processing provides a better gain when there are as many compute units working on the same task as possible, as inactive or idle units do not aid the overall computation, which would increase the time taken. By finding these inactive processing units, the algorithm can be optimised to reduce the amount of time a given processing unit has to wait for work to do.

Profiling OpenCL based software

Despite being an open standard, there is limited support for tools available to developers for profiling OpenCL software. It is non-trivial to use NVidia's software for profiling OpenCL programs, as NSight Compute and NSight Systems are more suited to CUDA and Nvidia GPUs AMD provides a software tool known as CodeXL, which is a part of the GPUOpen project SDKs, 2020. This tool allows for the profiling and debugging of OpenCL kernels, which is useful as it helps to optimise the kernels used. However it only works on AMD GPUs, where as our chosen framework, Tensorflow uses CUDA as its parallel backend.

OpenCL also has a built-in profiling tool which uses an event based system, however its functionality is limited. This is because the events only register at significant points in an OpenCL kernel lifecycle. Whilst it is useful for identifying a parallel function causing the bottleneck, it offers no further information on where a developer should begin looking to optimise. This means that it is more suitable for this project to use NVidia software for profiling our parallel code due to its links to CUDA.

2.3.3 Unit tests

A big problem with creating and improving software is that it is likely to break. This is especially true when refactoring to speed up the code. This is due to changes in underlying implementation creating side effects in the code. These side effects cause the code to break, and if left unchecked increases the technical debt, which accrues as a natural part of the software development life-cycle.

```
def boltzmannProbs(W, x, axis=0):
    """
    :param W: Input Weight Matrix
    :param x: Input Vector Matrix
    :param axis: Axis of contraction
    :return: activation probabilities
    """
    reward = tf.tensordot(W, x, axis)
    E_on  = tf.negative(reward)
    E_off = 0.0*E_on
    Q_on = tf.math.exp(-E_on)
    Q_off = tf.math.exp(-E_off)
    P_on = tf.math.divide(Q_on, tf.math.add(Q_on, Q_off))
    return P_on
```

Code Segment 2.4: Example unit to be tested

Unit tests are a method of both reducing technical debt and ensuring that behaviour remains the same. This is because the tests are used to verify behaviour after refactoring a unit. A unit can be considered any functional object, however we define a unit as a custom function in the code. An example of a unit can be seen in Code Segment 2.4.

A test is a wrapper function that provides known inputs to the unit and checks the output of the unit against a known output. The check is done via an assert statement. An assert statement fails if the values that the assertion is checking do not match. We define a test function as a singular concept on the unit being tested. This allows us to understand the nature of the unit for that concept. It also reduces confusion on what a given test is meant to do. This reduces technical debt as future developers are not having to figure out what the purpose of a given test is, which increases productivity. An example of a test can be seen in Code Segment 2.5

```
def test_rbm_small():
    input_weights = np.array([[.1, .2], [.3, .4]])
    input_vector = np.array([.5, .6])
    calculated_results = rbm.boltzmannProbs(input_weights,
   input_vector)
    actual_results = np.array([0.54239794, 0.5962827])
    assert_almost_equal(calculated_results, actual_results,
                            err_msg="[BoltzmannProbs] Calculated
   results are not equal to actual results to 7 decimal places.")
```

Code Segment 2.5: Example test for a unit

Test Driven Development and HC-UCPF

Test Driven Development is a method of developing software that focuses on writing tests before writing code. This allows the developer to write valid code the first time round, and does not need to spend long hours debugging the code. This works well for new projects. However our project involves refactoring an existing project. Thus we can not follow test driven development to the letter. But we can use the approach of test driven development as part of the refactoring process. This means we write our tests and ensure that the existing code passes them. This also has the benefit of helping to understand the purpose of the code.

By refactoring code in this manner, it helps with the process of debugging. This is due to being able to pass the system inputs to the tests, and seeing what is causing the error. Furthermore, from our definition of the unit test, we can obtain a hierarchical approach to debugging. This is due to each of our functions being units, and invoking other functional units within our system.

For example, the function `learn` in learnWeights.py[1] invokes the `boltzmannProbs` function in rbm.py[2]. If an error occurs during the learn function with respect to the `boltzmannProbs` function, we can pause execution of the learn function and run the current inputs through our tests to find where the erroneous behaviour occurs. Once we understand the erroneous behaviour, it can be considered trivial to fix it. This would then allow successive invocations of the system to run without encountering the error.

2.3.4 Refactoring with Unit Tests

Unit tests are traditionally used for the creation of software, as they are designed to help a developer create software. This is because you can define the expected behaviour of a function and work to that definition. However they can also be used

[1]Found at `https://github.com/A-Yakkus/hclearn/blob/msc2020/learnWeights.py#L16`
[2]Found at `https://github.com/A-Yakkus/hclearn/blob/msc2020/rbm.py#L9`

in conjunction with profiling for speeding up known functions via refactoring. This is because we can write the tests for a function that is causing the bottleneck from profiling. We can then develop a new version of the function with Tensorflow, using these tests to preserve behaviour.

A prominent figure in the field of refactoring is Martin Fowler. Fowler describes unit testing as being situational, with a unit referring to an agreed upon identifier, such as a function, class or object, and a test referring to a fast, low-level implementation of the unit, focusing on a small part of the system (Fowler, 2014).

Fowler suggests a hat-based approach to refactoring. This means that as a developer, you would wear different metaphorical hats depending on the job. Whilst wearing a particular hat, you purely focus on that job. This reduces the chance of getting confused on what you are doing when working on the code.

For our project, we are refactoring the model to allow for parallel execution of the learning process via unit testing. This will also have the effect of making the learning easier to understand moving forward. Hence we are reducing the technical debt of the project. This allows us to create a number of hats for the different tasks, as follows:

- A hat for the creation of unit tests.
- A hat for the development of parallel functions using Tensorflow.
- A hat for managing the integration of the parallel functions into the overall system.

These hats can help guide the development of the project. By developing the project in this way, we can control the scope of the refactoring, and keep it focused on the learning process, rather than the setup or evaluation.

Using the unit tests as a base for our refactoring is useful as it allows for faster system analysis in the event of a crash. This is because the stack trace displayed when a crash occurs usually identifies the line that caused the crash. This relies on the developer understanding the system to know how to fix it.

However our hierarchical approach to creating the unit tests reduces the amount of knowledge required. This is because the inputs can be retrieved from a breakpoint in the debugging process, and then transferred to being inputs to a test script. This test script then runs the unit tests on all functions and nested functions within that get invoked from the crash causing line. Thus the overall time needed to fix the crash gets reduced as the unit tests identify how the crash is caused, making it easier to fix. Furthermore, by reducing the amount of knowledge needed to understand the system at the onset of a project in this manner, we are able to curb the growth of future technical debt that the system may obtain.

Chapter 3: Methods, Experiments and results

As we are using test driven development in conjunction with profiling to guide the refactoring, we present our general methodology for refactoring and then each optimisation technique. Alongside each optimisation, we provide results and analysis on what each optimisation does to the model. Finally, we analyse the profiling and verify the model after all optimisations.

Understanding the model for refactoring

Before we can perform good refactoring of the model, we first need to understand how the model works, and what can be done to limit the variation in overall results that are seen. We discuss how the computational implementation of the model works, and how randomness can affect results, alongside minimising the effects of randomness.

The Unitary Coherent Particle filter of Hippocampus (UCPF-HC) model was initially developed in 2010, when serial computing was prevalent. The learning method, contrastive divergence (Hinton, Osindero and Teh, 2006) is a primarily serial processing method, with each successive sample requiring the previous one. This means that there are limited points of parallelisation in the learning process.

There are two key operations in our implementation of the learning process that should be parallelised, the calculation of individual probabilities for each input type to the model (biases, recurrent values, odometry and sensors) and the fusion of these probabilities into a singular joint probability table. These operations can be considered parallel as each node of the model is independent of the others for these operations, meaning that a parallel worker can be applied to each node for the calculation, which should reduce the overall time to learn.

Randomness in the UCPF-HC

The UCPF-HC model uses randomness in two points of the execution of the system, in building the representation of the world and determining which neurons are active in the hidden state of the Temporal Restricted Boltzmann Machine (TRBM).

Pseudo random number generators (pRNGs) are used to create the random numbers for the model's learning process. However many pRNGs generate random numbers using a serial algorithm. This is a problem in parallel computing as each calculation thread operates independently. This means that if a thread requires access to a random number, it may retrieve a different value depending on the order the threads access the pRNG. This would then cause the output of anything depending on the random function to be different, which breaks our concept of unit testing. Seeding the pRNG reduces the effect of the random numbers as it allows for a predictable sequence of numbers to be used. However the access to the pRNG is still relatively random from the operating system scheduler.

Nvidia is currently working on a framework determinism project Nvidia, 2020. This project aims to make parallel random number generation deterministic. This is useful as it gives complete control over the usage of the random number generator. However the project is currently in the alpha/beta stage of development. This indicates that there is not full, stable support for dependents of the project. Furthermore, the main project does not appear to have been updated in the last 10 months, which suggests that the project has been dropped for the time being.

3.1 Process of refactoring

The first step of our refactoring and optimisation process is to identify the best function to optimise. This is done by profiling the code using cProfile, and using a chart of most time taken for our functions. After this we traverse down the call stack to find the deepest function, i.e. the function that does not call any other function from our program. We then implement unit tests for this function, considering the data types that we expect to see, such as real floating point numbers, data types that we are unlikely to see, such as imaginary numbers and integers, and data types that we shouldn't see, such as strings and booleans. This allows us to understand the behaviour of the code under different situations. As the goal of the project is to maintain/improve execution time, we also consider cases of small inputs, such as a 2x2 matrix and a 1x10 vector and large inputs, such as a 1000x1000 matrix and a 1x1000 vector. This will allow us to compare how well the different implementations handle horizontal scaling in isolation.

Once the unit tests, examples of which can be seen in Code Segment 3.1 have been written, we implement our parallelised version using the Tensorflow library. We

can then use our unit tests to ensure that our implementation matches the current implementation. After verifying our implementation, we can then integrate it into the original system. Once integrated, we can repeat the process to further improve the overall system. These functions would come from any function used in the learning process, as it will allow us to move as much computation onto the GPU as possible, which would speed up the overall learning process.

```
File: cffun.py
Function: Lag
def test_thousand_by_thousand_hundred(self):
        results = cffun.lag(self.initial_thousand_by_thousand, 100)
        actual = np.vstack((np.repeat([self.
   initial_thousand_by_thousand[0, :]], 100, 0), self.
   initial_thousand_by_thousand))[:-100, :]
        assert_array_equal(results, actual, "[Lag] When lagging by
   one on a 2D array with second dim=1, result should be the same")

Function: Invsig
def test_vector_10(self):
        calculated_results = cffun.invsig(self.vector_10)
        actual_results = -np.log((1./self.vector_10)-1)
        if type(calculated_results) is tf.Tensor:
            calculated_results = calculated_results.numpy()
        assert_almost_equal(calculated_results, actual_results)

File: rbm.py
Function: Hardthreshold
def test_two_by_two(self):
        calculated_results = rbm.hardThreshold(self.initial_2)
        actual_results = np.array([[0, 1], [1, 0]])
        assert_almost_equal(calculated_results, actual_results,
                            err_msg="[HardThreshold] Calculated
   results are not equal to actual results to 7 decimal places.")
```

Code Segment 3.1: examples of unit tests for different functions, labelled by the file and function which it refers to.

3.2 Evaluating the refactoring process

The effectiveness of our refactoring will be evaluated by the average runtime of the system. We do this by timing how long 10 consecutive invocations take and calculating an average time taken. An average time allows us to account for operating system schedulers as well as transfer time of data between the CPU and GPU. This time will then be compared against the pure CPU implementation, and we should expect to see some reduction in time taken.

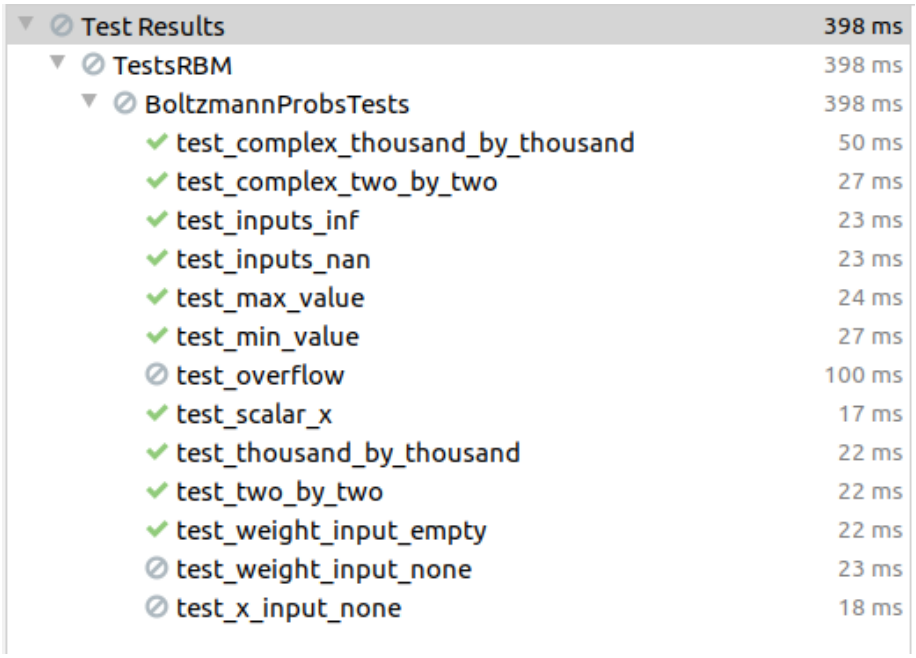

Figure 3.1: Unit test results for the NumPy version on CPU of the `boltzmannProbs` function.

We obtain the time taken by using the unix `time` command. This provides three different measures of time taken, wall clock or "real" time, user time and system time. Wall clock time refers to the difference between the unix timestamp when the program starts and when the program exits. User time and system time (CPU time) accumulate based on the actual amount of time the CPU is being used for the program being timed.

We will be using wall clock time as our measurement. This is to provide an understanding of how long the process takes as with respect to the real world. We could use CPU time, which only tracks how long the process takes on the CPU only. This is not useful to our project, as more operations will take place on the GPU, which is not considered as part of CPU time. This would distort the results, showing that there is a significant improvement to the model, whereas in reality this may not be the case. In contrast, as the wall clock time can be considered consistent relative to changes within the software, this reduces the effect of the distortion caused by executing operations on the GPU over the CPU.

3.3 Unit Test Results

After writing the unit tests we can use tools to automatically run them. This allows us to monitor and manage the unit tests.

The initial tests were written with respect to the existing code, and to show a variety

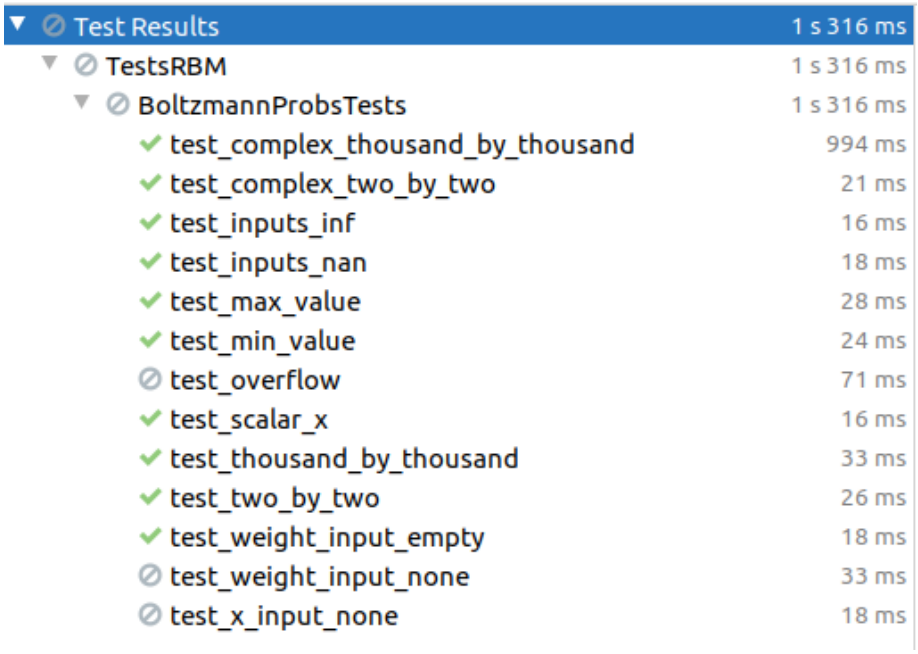

Figure 3.2: Unit test results for the Tensorflow version on GPU of the `boltzmannProbs` function.

of behaviour for the function. As such the tests will always pass, as seen by the green ticks in Figure 3.1 The tests showing as skipped (grey circle with line through it) are tests that are expected to fail. This helps to manage the behaviour of the model.

As each test focuses on a single aspect, we can compare between the different versions of the model. For example, the first test executed is the test on an array of large complex numbers. This case is unlikely to occur, as it is not normally encountered in regular use. On the GPU, this test takes approximately 1000ms, as seen in Figure 3.2. In contrast, the test only takes 50ms on the CPU, seen in Figure 3.1. This is a 20x slowdown in the GPU version, which is likely due to the effect of initialising Tensorflow. Subsequent tests then show similar times to the NumPy versions, with several tests on the GPU matching or improving upon the NumPy equivalent times.

The test that shows the most interesting behaviour is the 1000 by 1000 test, which seems to indicate that the GPU version is slower by approximately 50%. This is likely an anomalous result, as the GPU version matches the CPU version on most of the other tests.

3.4 Function Timing

TEST ID	PARALLEL FUNCTION IMPLEMENTATION DETAILS	SIZE(of CA3)	TIME(s)	MAZE	TRAIN TYPE
0	Initial Implementation	86	83.323	PLUS	FULL
1	boltzmannprobs	86	407.743	PLUS	FULL
2	boltzmannprobs and addBias	86	410.667	PLUS	FULL
3	boltzmannprobs, addBias and hardThresh	86	408.420	PLUS	FULL
4	addBias and hardThresh	86	89.255	PLUS	FULL
5	addBias	86	90.315	PLUS	FULL
6	hardThresh	86	85.431	PLUS	FULL
7	trainPriorBias and hardThresh	86	85.334	PLUS	FULL
8	trainPriorBias, invsig and hardThresh	86	85.753	PLUS	FULL
9	invsig and hardThresh	86	84.903	PLUS	FULL
10	invsig	86	86.566	PLUS	FULL
11	trainPriorBias and invsig	86	86.791	PLUS	FULL
12	trainPriorBias	86	86.705	PLUS	FULL
13	outer convienience function	86	284.001	PLUS	FULL
14	outer and boltzmannProbs	86	428.402	PLUS	FULL
15	outer, boltzmannProbs and Tensorflow random	86	423.419	PLUS	FULL
16	outer, highly optimised boltzmannProbs and Tensorflow random	86	352.254	PLUS	FULL
17	outer, decorated, highly optimised boltzmannProbs and Tensorflow random	86	335.582	PLUS	FULL

Table 3.1: Table of results comparing different functions as we parallelise the learning process.[1]

Table 3.1 shows how parallelising a given set of functions affects the runtime of the system after integration. The test id column refers to which test is being run. The second column is defining which functions have been parallelised, with test 0, being CPU only functions. The size column refers to the size of the CA3 sub-field, which is kept at 86 to help maintain consistency across different implementations. The time column is the average wall clock time it takes to run the system. An average time is used to account for differences at the operating system level, such as interrupts and transfer overhead, which can occur at different times. The final two columns give insight into the environment the model is learning (see Figure 3.3, and the different methods of training the model. These training methods include using the DG only, using handset weights only or using random weights only. We opted to use all three training types to show that our optimisations can apply to each method.

It can be seen that there are two distinct bands of time, depending on the parallel functions integrated. This is likely due to the number of calls to the `boltzmannProbs` and outer functions, which results in multiple data transfers that is expensive.

Test IDs 16 and 17 show a further optimised version of the `boltzmannProbs` function. This is because the initial optimised `boltzmannProbs` function was a translation of the original `boltzmannProbs` function. However Tensorflow does not have a replacement for the NumPy dot function. This means that the multiplication of a matrix and scalar value does not work in Tensorflow. To get around this limitation, an if statement for type checking was used. But this added several unnecessary operations

[1]Results were gathered on a AMD FX8320 CPU (3.5GHz clock speed), 16GB 1600MHz Quad channel RAM and ASUS GT710-SL-2GD5 GPU (Uses Nvidia Geforce GT710 technology)

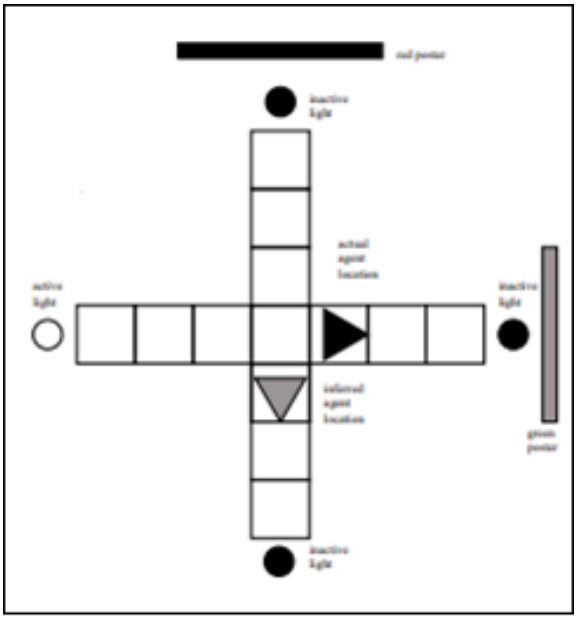

Figure 3.3: The plus maze environment the model is learning, from Fox and Prescott, 2010a

to the code. Thus a more optimised form of the function, which primarily depends on the `tf.tensordot` function is used, with inputs being forced into matrices before being passed to the function. The decorator applied to the `boltzmannProbs` function in test ID 17 is `tf.function`. This provided the optimisation of compiling the function into a data flow graph instead of executable Python code. This resulted in the model being faster, and the version used for further analysis.

3.5 Artificially expanding the size of the CA3 sub-field

As the model is derived from the classical view of the hippocampus, the CA3 sub-field is represented by a TRBM. The size of the CA3 sub-field is dependent on the environment being learned. For example, the plus maze environment, Figure 3.3, is made of 13 segments. This gives us 13 nodes for position, and 52 nodes for the combination of position and direction headed. In addition, the model allows for 4 nodes of sensory data, as well as the combination of sensor and direction which total 16. Finally, we add in a bias node, which gives a base size of the CA3 of 86.

From how the CA3 sub-field is created, it provides two main options for increasing it's default size: changing the environment the model learns and increasing the number of biases in the model. Changing the environment would be impractical as multiple new environments that have different sizes would need to be created to fully explore how the time to learn changes as the CA3 sub-field increases in size. On the other

hand, increasing the number of biases in the model is trivial, and can be used to provide a comparison of how the time to learn is affected by multiple size changes to the CA3 sub-field.

We expect to see that the more nodes we add to the CA3, the time taken to train increases at a constant rate. On the CPU version, we would expect to see a steep increase in time taken, whereas the GPU version would be a more gradual increase.

A problem that occurs when increasing the size of the CA3 sub-field is the increase in memory usage. This is because Tensorflow converts from NumPy floating point arrays and Python floating point numbers to a 64 bit floating point tensor of the same size. This is not suitable for GPU processing. This is because GPU's have the best efficiency when working on 32 bit floating point numbers. Whilst this change does reduce the overall precision of the model, for the purposes of this experiment, it can be considered to have a negligible effect on the results.

3.5.1 Results from increasing the size of the CA3 sub-field

As the number of neurons increases, the amount of calculations increases. Naturally this increases the amount of time the system needs to learn. As can be seen from Table 3.2, increasing the size of the CA3 subfield also increases the time required to train the system.

Number of Nodes in CA3	Time Taken CPU (s)	Time Taken GPU (s)
86	83.3	335.6
1086	770.3	441.3
2086	2437.2	625.7
3086	4653.8	896.9
4086	7391.2	1231.5
5086	10337.4	1642.0
6086	14137.6	2154.4
7086	19531.4	2739.1
8086	24685.6	3423.7
9086	27058.8	4155.0

Table 3.2: Time taken to run the training process for different sizes of an artificially inflated CA3 using the model variations from test 0 and test 17.[2]

As we linearly increase the number of nodes from 3086, it requires approximately 3600 seconds (1 hour) more time. In contrast, using the parallelised Tensorflow version, we only require approximately 420 seconds (7 minutes) more time for increased nodes. This is a speed up of roughly 10 times for extra nodes. These times were recorded on

the implementations of Test ID 0 for the CPU version and Test ID 17 implementation for the GPU version.

3.6 Optimisation of data type casting

Relating to the data type optimisation, data type casting is an expensive operation. This is due to the need to reassign memory to match the new data type. This does not affect the conversion from NumPy arrays to Tensorflow tensors as they are initialised as 32 bit floats. However the model uses a comparison of probability against random values to create a new hidden state representation after fusing the probabilities from the different inputs. This creates a boolean tensor. NumPy is able to utilise Python's weak typing to use a boolean tensor as a numerical representation, but Tensorflow uses strong typing. This causes us to need to cast the boolean values to float values. As this is happening multiple times in every step of the learning process, it provides a candidate for optimisation.

3.6.1 Results from optimising the casting operation

One method for optimising the casting operation is to use the `tf.where` function. This creates a tensor of two values using a conditional statement to determine which value to use. This is useful as the `tf.where` function performs the same operation as casting. Using the Timeit module, we can test whether this operation is faster than casting.

Number of Nodes	tf.cast	tf.where
10	62.073	43.839
100	62.091	43.934
1000	62.208	44.035
10000	61.614	44.288

Table 3.3: Timing 100000 invocations of each function over different numbers of nodes. The times reported are cumulative of all 100000 invocations.

From Table 3.3, it can be seen that there is a noticeable speed up by using `tf.where` across different numbers of nodes. This implies that implementing this optimisation into the model will reduce the overall time taken to learn. After implementing the optimisation into the model, we get an average runtime of 332.2 seconds. This is a minor improvement over the implementation of test ID 17, however it is an improvement thus we will be using this optimisation in future experiments.

[2]See Footnote 1 for system details.

3.7 Optimisation of the error function

We can use line profiling to identify other areas of optimisation. An example of this is the error function, which takes up 20% of the execution time. This makes it suitable candidate for optimisation. Looking at the code in Code Segment 3.2, lines 2, 3, and 5 are element-wise operations, meaning that the parallelisation of these operations are simple, as Tensorflow tensors can already handle them. The interesting line in this code is line 4. This is because the Python keyword `sum` performs a serial operation, which is inherently slow, and has no direct counterpart in Tensorflow. However, there is a similar function, `tf.reduce_sum` which exhibits the same behaviour in a parallel manner.

```
1 def err(probabilities, hidden_state):
2     difference = probabilities-hidden_state
3     squared_difference = difference ** 2
4     sum_difference = sum(squared_difference)
5     error = sum_difference / hidden_state.shape[0]
6     return error
```

Code Segment 3.2: Error function split into each operation

3.7.1 Results from optimising the error function

As seen in Table 3.4, `tf.reduce_sum` maintains a relatively constant time regardless of the number of nodes used. This indicates that the optimisation is useful for reducing the overall learn time of the model, when comparing against a pure CPU implementation.

Number of Nodes	sum (s)	tf.reduce_sum (s)
10	0.183	0.018
100	1.496	0.011
1000	14.652	0.010
10000	144.899	0.011

Table 3.4: Timing 100 invocations of the named functions across different numbers of nodes. The times reported are cumulative of the 100 invocations.

Implementing this optimisation into the model using 86 nodes in the CA3, we obtain an average runtime of 331.1 seconds. This is another minor improvement alongside the casting optimisation above, indicating that there is still a bottleneck in the code. However, when the model is trained to learn a larger area, with more nodes, this optimisation will be beneficial.

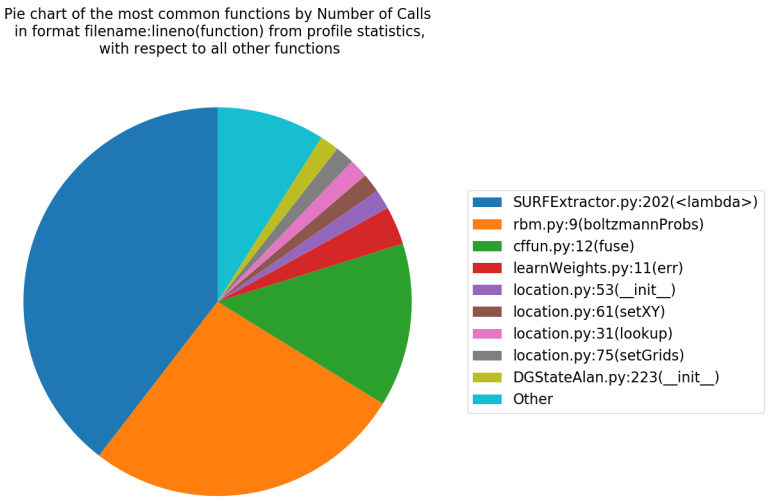

Figure 3.4: Number of calls for the CPU implementation of the model.

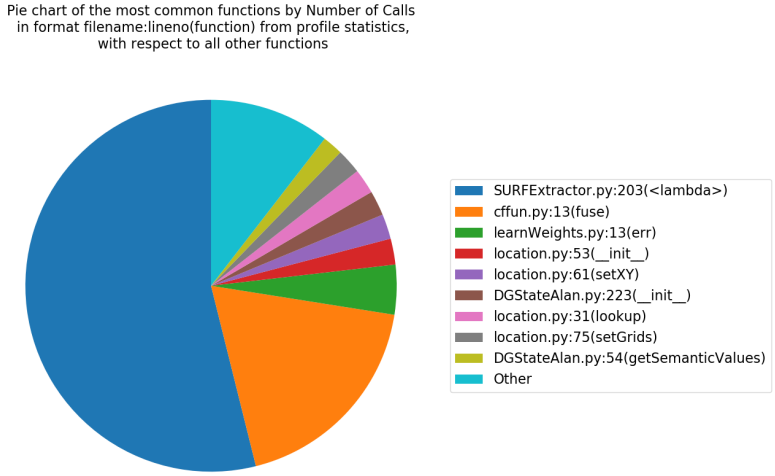

Figure 3.5: Number of calls for the GPU implementation of the model.

3.8 Profiling Results

3.8.1 Number of calls

From Figure 3.4 we can see that the most called function, a lambda in the SURFExtractor module is part of the setup process, meaning the relevant functions are `boltzmannProbs` in the rbm module, the `fuse` function from the cffun module and the `err` function in the learn weights module.

In contrast, profiling the Tensorflow version (Figure 3.5) shows the lambda function from SURFExtractor taking a larger proportion of the total number of calls. This is likely due to the usage of the `tf.function` decorator optimisation. The decorator

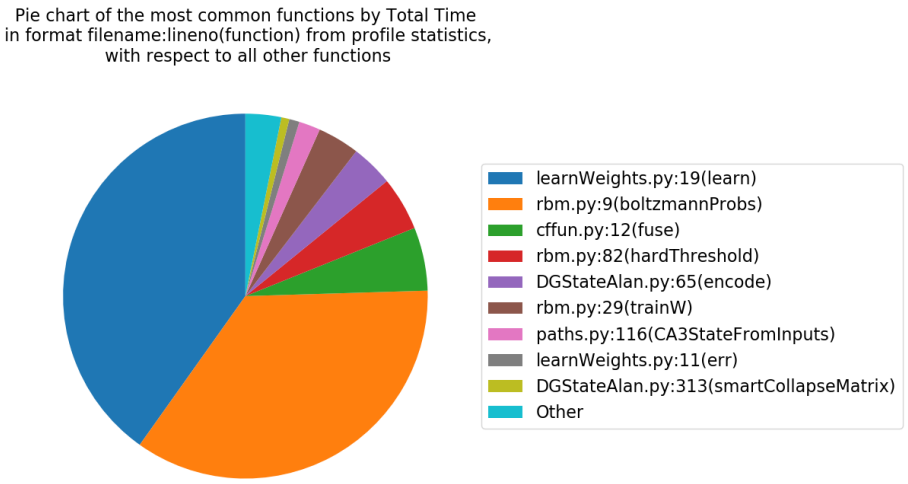

Figure 3.6: Total time for function calls for the CPU implementation of the model.

converts the function into an optimised Tensorflow data call graph, which is loaded into the memory of the GPU at runtime.

Tensorflow processes the calls to the function such that the profiler only records the first time each call to `boltzmannProbs` is hit, as opposed to everytime. The `cffun#fuse` function has approximately the same amount of calls, and can probably be reduced by applying the `tf.function` decorator. Finally the `learnWeights#err` function has a constant number of calls across both implementations as it is not affected by the randomness of the model.

3.8.2 Total time

The more interesting profiling results come from the total time spent executing the function, excluding calls to other functions. Initially the main functions that take the most time are the `learn` function of the `learnWeights` module, the `boltzmannProbs` function of the `rbm` module and the `fuse` function of `cffun` module as seen in Figure 3.6.

After applying the parallel refactoring, the `learnWeights#learn` function takes a larger proportion of the overall time taken, as seen in Figure 3.7. The `cffun#fuse` function also has an increase in the proportion of time taken. However, given that the overall time taken increases with a small amount of nodes, it indicates that there may be an underlying algorithmic problem which causes this behaviour.

In addition, the `boltzmannProbs` function appears to have a large reduction in time taken. However this is likely due to the `tf.function` decorator function mapping the original function to a data call graph that runs purely on the GPU, which the Python profiler does not have access to. This can be seen when using the line profiler on the `boltzmannProbs` function.

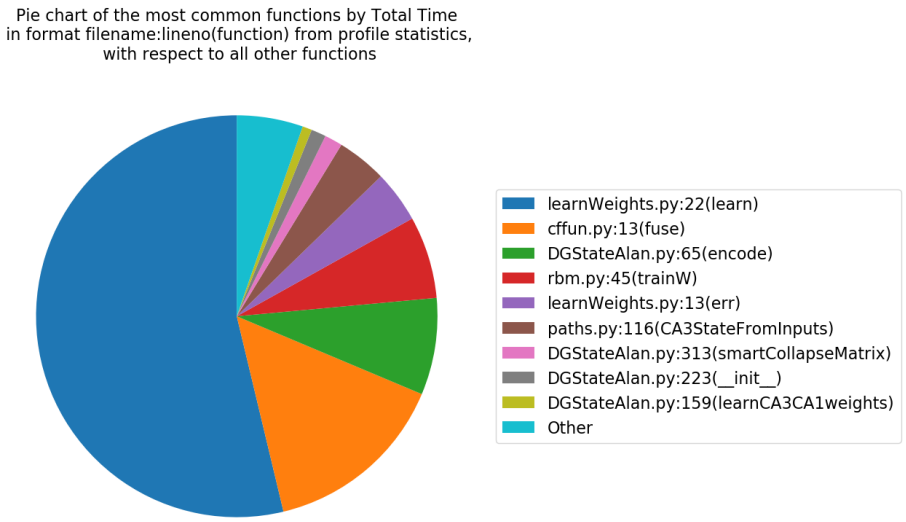

Figure 3.7: Total time for function calls for the GPU implementation of the model.

Code Segment 3.3 shows the results of profiling the `boltzmannProbs` function line by line. It can be seen that the main bottleneck is the multiplication of the inputs and weights, taking approximately 60% of the overall time. This is likely due to the asynchronous nature of matrix multiplication, which is applied across a given dimension of the inputs.

```
Total time: 237.186 s
Function: boltzmannProbs at line 9

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     9                                           @profile
    10                                           #@tf.function
    11                                           def boltzmannProbs(
   W, x, axis=0):      # RETURNS THE PROBABILITY OF A NODE BEING ON
    12    252240  143312866.0    568.2     60.4      mult = tf.
   tensordot(W, x, axis)
    13    252240   12193753.0     48.3      5.1      squeezed = tf.
   squeeze(mult)
    14    252240    6917896.0     27.4      2.9      E_on  = -
   squeezed       #penalty is the negative of the reward (just to
   make it look like energy
    15    252240   20604441.0     81.7      8.7      E_off = 0.0*
   E_on
    16    252240   13057317.0     51.8      5.5      Q_on = tf.math.
   exp(-E_on)       #as energy is negated, we have e^(reward)
    17    252240   12601996.0     50.0      5.3      Q_off = tf.math
   .exp(-E_off)
    18    252240   28258157.0    112.0     11.9      P_on = Q_on / (
   Q_on + Q_off)
    19    252240     239234.0      0.9      0.1      return P_on
```

Code Segment 3.3: Line by line profiling of the rbm#boltzmannProbs function in eager execution mode

3.9 Verification of model accuracy

The results outlined above only consider the learning process. However for our optimisations to be considered successful, we need to compare the accuracy of the initial serial implementation of the model with the current parallelised implementation.

Version	Learned Weights	Handset Weights	Random Weights
CPU	63.70%	62.33%	19.37%
GPU	41.93%	67.27%	19.37%

Table 3.5: Table comparing the accuracy of the model with respect to a 3000 step walk in the plus maze for each type of learning method, using handset weights, learned weights and random weights.

Table 3.5 shows the accuracy of the model for both the original CPU implementation and our parallelised implementation. We consider 3 types of weights, weights learned from our learning process, handset weights of the model and random initialisation

of weights. For the random initialisation of weights there is no change in accuracy. This is due to the seed for initialising the weights being the same across both models.

With the handset weights, our implementation shows an overall increase of 5% in Accuracy. This suggests that the parallelised model obtains a more accurate picture of it's environment when given a good starting weight value for each area of the model. However there is a loss of 20% accuracy when using learned weights from the learning process. This implies that there is an error in our learning process that affects the overall accuracy.

The next step from this is to add unit tests to both the learning and the evaluation process. These tests would be targeted to capture the behaviour which causes the learned accuracy to be reduced. This would allow the problem to be investigated and fixed, and would improve the accuracy to being approximately the same as the initial, CPU version.

Chapter 4: Discussions and Conclusions

Our results indicate that we have achieved our goal, especially when the size of the CA3 is large, which makes the model more suitable for real world use. However the environment is relatively small, and the inflation of the CA3 sub-field can be considered artificial. This means that there may be a small performance improvement for a larger environment. This can be checked by using the current parallel implementation in a new environment which covers a larger and more complex environment in future work.

Next, we assumed that Tensorflow was the best option for parallelisation. This is because of its popularity and prominence as a machine learning library. However, as we outline in Section 2.2.2, there are many libraries for Python which handle parallelisation and scaling. As such we may be able to obtain better results from a different library. This is because Tensorflow has initialisation overhead on first call, which can slow down the learning process, whereas other libraries will have different points of initialisation overhead, or does not load unnecessary modules, e.g. Keras.

Profiling the code showed that the learning process was slow. This made it the best segment to optimise. However our results indicate that the optimisation is only effective at scale. We can assume that the problem lies within the matrix multiplication of each type of sensory/weight data in the model, (data from sensors, odometry data, recurrent neurons in the CA3 and the biases) when looking at the current environment. This suggests that further work into optimising this multiplication will heavily reduce the overall time to learn, given that particular multiplication occurs 250,000 times.

Furthermore, our secondary metric of model accuracy shows that our refactoring of the model is less than satisfactory. This is because of the dramatic loss of accuracy of the learned weights. However, our refactored model still performs twice as well as randomly selecting weights. The model is also influenced by randomness, which may have played a part in how well the model learns, considering only 10 epochs are used during the training process. This indicates that further work needs to be done

to understand how the model reacts to randomness, and how that affects the overall accuracy of the model.

Whilst we attempted to account for as many variables as possible within the optimisation process, the overall randomness of the model causes variations within results to occur. An example of this is the model randomly choosing to account for observations and recurrent values in the contrastive divergence learning. This can be managed by setting a seed, however the asynchronous nature of parallel computing means that different threads can retrieve the next number in the random number generation in a different order with successive invocations. Thus we chose to allow for the randomness to affect the learning process to provide us with a set of results that can be expected in a real world environment.

In conclusion, our results show that our current optimisations only have an effect on the model at scale. This is useful as real world environments are larger than the plus maze environment the model currently learns. Finally, we have laid the foundations for future work to build upon our current refactoring of the model to Tensorflow using unit tests.

References

Al-Rfou, R. et al. (2016). 'Theano: A Python framework for fast computation of mathematical expressions'. In: *CoRR* abs/1605.02688. arXiv: 1605.02688. URL: http://arxiv.org/abs/1605.02688.

Andersen, P. et al. (2006). *The hippocampus book*. Oxford university press.

Bahrampour, S. et al. (2015). 'Comparative Study of Caffe, Neon, Theano, and Torch for Deep Learning'. In: *CoRR* abs/1511.06435. arXiv: 1511.06435. URL: http://arxiv.org/abs/1511.06435.

Bengio, Y. (2017). *MILA and the future of Theano*. Online; Accessed 2020-07-20. URL: https://groups.google.com/forum/#!msg/theano-users/7Poq8BZutbY/rNCIfvAEAwAJ.

Cajal, S. R. y (1911). *Histologie du Système nerveux de l'Homme et des Vertébrés*.

clMathLibraries (2020). *clBLAS*. Online; Accessed 2020-07-26. URL: https://github.com/clMathLibraries/clBLAS.

Dhariwal, P. et al. (2020). *Jukebox: A Generative Model for Music*. arXiv: 2005.00341 [eess.AS].

Fowler, M. (2014). *Unit Test*. Online; Accessed: 2020-07-26. URL: https://martinfowler.com/bliki/UnitTest.html.

Fox, C. and Prescott, T. (2010a). 'Hippocampus as Unitary Coherent Particle Filter'. In: *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pp. 1–8.

– (2010b). 'Learning in a Unitary Coherent Hippocampus'. In: *Artificial Neural Networks (ICANN)*.

Hinton, G. E., Osindero, S. and Teh, Y.-W. (2006). 'A fast learning algorithm for deep belief nets'. In: *Neural computation* 18.(7), pp. 1527–1554.

insights, M. and strategy (2016). 'AMDs ROCm Software Helps Accelerate HPC and Deep Learning'. In: (Accessed: 14 August 2020). URL: https://moorinsightsstrategy.com/wp-content/uploads/2016/11/AMDs-ROCm-Software-Helps-Accelerate-HPC-and-Deep-Learning-by-Moor-Insights-and-Strategy.pdf.

Klöckner, A. et al. (2012). 'PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation'. In: *Parallel Computing* 38.3, pp. 157–174. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.09.001.

Looie496 (2008). *Hippocampus anatomy*. Online; Accessed 2020-08-17. URL: https://en.wikipedia.org/wiki/File:CajalHippocampus_(modified).png.

Nugteren, C. (2017). 'CLBlast: A Tuned OpenCL BLAS Library'. In: *CoRR* abs/1705.05249. arXiv: 1705.05249. URL: http://arxiv.org/abs/1705.05249.

Nvidia (2020). *Framework Determinism*. Online; Accessed 2020-08-16. URL: https://github.com/NVIDIA/framework-determinism.

Nvidia C.U.D.A. (2020). *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Online; accessed 07 July 2020].

Okuta, R. et al. (2017). 'CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations'. In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. URL: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

Paszke, A. et al. (2019). 'Pytorch: An imperative style, high-performance deep learning library'. In: *Advances in neural information processing systems*, pp. 8026–8037.

Saul, A., Prescott, T. and Fox, C. (2011). 'Scaling up a Boltzmann machine model of hippocampus with visual features for mobile robots'. In: *2011 IEEE International Conference on Robotics and Biomimetics*, pp. 835–840.

SDKs, G. L.
bibinitperiod (2020). *GPUOpen Libraries & SDKs*. Online; Accessed 2020-07-26. URL: https://github.com/GPUOpen-LibrariesAndSDKs.

Tensorflow (2020). *Tensorflow Tutorials*. Online; Accessed 2020-08-11. URL: https://www.tensorflow.org/tutorials.

Thies, J. et al. (2020). 'Neural Voice Puppetry: Audio-driven Facial Reenactment'. In: *ECCV 2020*.

Wang, S. et al. (2020). 'A Fully Automatic Deep Learning System for COVID-19 Diagnostic and Prognostic Analysis'. In: *European Respiratory Journal*. ISSN: 0903-1936. DOI: 10.1183/13993003.00775-2020. eprint: https://erj.ersjournals.com/content/early/2020/05/19/13993003.00775-2020.full.pdf. URL: https://erj.ersjournals.com/content/early/2020/05/19/13993003.00775-2020.

Appendix A: Additional assessment information

We have written this report in the form of a scientific paper as we plan to publish it. However the MSc Research Project assessment criteria requires additional information that is not usually included in publications. As such, we include this information below.

A.1 Aims and objectives

The aim of the project is to use Tensorflow to parallelise a model of the hippocampus to make it more suitable for real-time/real world use. To do this we will need to complete the following objectives:

- Write unit tests to understand current system behaviour
- Ensure the initial implementation passes unit tests.
- Write Tensorflow-based Python code that emulates current system behaviour.
- Verify our Tensorflow code works as expected using the unit tests.
- Time the system using different combinations of the original code and parallel code,
- Profile the final system and compare it to our original profiling results.

A.2 Tools and Toolsets

As a software engineering project, we will need to use tools outside those covered above to manage the development of the model.

A.2.1 Development Environment

The model is developed on Linux, as it would be a similar operating environment that would be used in the real world. We use Jetbrains Pycharm as our development environment as it provides quality of life features such as code completion and inspection. It also has integration with Git which makes it easier to see which files have been modified. This helps in managing which functions have been integrated into the system as a whole. It also allows for the management of virtual Python environments, which separate Python libraries to reduce confusion on which libraries to use. Below is a list of libraries that need to be installed into the environment in order to obtain a minimum working environment.

- Pip version: 20.1.1
- NumPy version: 1.18.4
- Matplotlib version: 3.2.1
- Pyflann3 version: 1.8.4.1
- Opencv-contrib-python-nonfree version: 4.1.1.1
- Tensorflow version: 2.2.0

A.2.2 Git and Github

Git is a version control tool. It records changes to files in a tree based manner. This is useful for our project as it provides a method to return to a previous model state quickly and easily. This allows for different paths that the model can be developed in to be explored, which makes overall development faster. In addition, it also acts as a backup mechanism, by regularly pushing updates to repository sites like Github, as it allows the model to be restored at any time in the event of hard drive failure or loss of data.

A.2.3 Latex and Overleaf

Latex is a typesetting system. This allows it to have a singular source, but provide different document layouts. This makes it easier to convert the report to different formats for publication. There are multiple environments for processing latex documents, such as Overleaf, TexLive and MikTex. We chose Overleaf as it is an online environment, which suggests that there is some form of redundancy measures in place. This is useful as it means that if a local drive was to fail, the report would not

be lost. In addition, an online environment allows for easy collaboration with others. This makes it easier to work with my supervisor and proofreaders, as discussions on minor changes can be made quickly and easily. Another reason for choosing Overleaf instead of Texlive and MikTex is that it is a managed system. This means that extra packages, such as table/figure rendering and placement are instantly available. Whilst these packages can be installed into a TexLive or MikTex environment, it would slow down the creation of the report. Finally, in order to convert latex source to a suitable document for submission, it is customary to compile the same source code multiple times to fix intradocument referencing. This can be done in the other environments, however Overleaf provides a partially parsed log when compilation fails, which makes it easier to identify the problem and fix it as opposed to the other environments which would require identifying the problem from the logs.

A.3 Project Management

When developing software, there are multiple methodologies that can be applied, such as Agile and Waterfall. However our project revolves around refactoring an initial software package. This makes the Waterfall methodology unsuitable for the project, as the requirements are subject to change after each optimisation after refactoring. Agile practices, on the other hand are more suitable for our project. This is because the intermediary goals, such as which function to optimise next changes after each build of the system. This allows us to move fluidly through the development of parallel versions of the function, with no strict oversight on what should be done next.

As test driven development uses short software cycles to develop functions, it works well with an Agile approach. Furthermore, test driven development is suitable for refactoring as it allows us to manage deterministic behaviour easily. This means that if a function does not pass the tests, then it enables the developer to revisit the function and work out where the error that causes the test to fail occurs. Test driven development has been useful for this project in ensuring that behaviour is correct of our refactored functions.

Risk Matrix

When doing a software engineering project, there are risks that can occur during the project. We display these risks in Table A.1.

Risk	Severity Level	Impact	Likelihood	Mitigation
Hardware failure	4	4	1	Replace failed components of the machine with the same component
Data Loss	3	3	2	Regularly back up data to multiple external sources
Deviation from main optimisation points	1	4	3	Focus mainly on the big optimisations, and return to smaller optimisations at a future point in time.

Table A.1: Risk matrix for project.

A.4 Reflection

This project has been an enjoyable process for me. It has allowed me to expand my knowledge of different machine learning models and techniques. I usually use an iterative Waterfall approach to software development, using unit tests has been a welcome change of pace. It made the overall development of the project a lot smoother as it helped me understand the purpose of each function used in the learning process, and how they interact between each other.

The overall process of the development of the parallel version was smooth. This was due to the unit tests making it easier to see where problems were and how they may affect future model development over different iterations. By using unit tests in this project, I have found that there are more benefits to using unit tests than not using them, which I will be using in future projects.

When considering what did not go well, it was mainly the integration of each function into the overall system. This is because of the difference between Tensorflow and NumPy with different implementations of functions, such as the dot product. This caused a variety of solutions to be created to manage this, however they all had some variation that means they may not have been optimal. Another problem from the project is that the speed up seen in experimental results is not as expected, with approximately 2x speed up occurring at 1000+ neurons, and approximately 7x speed up at 3000+ neurons, whereas we were expecting to see around a 100x speed up with larger network sizes.

In comparison to my bachelors project, I feel this project has been more successful. This was likely due to the extra time that could be dedicated to the project, without other assignments to worry about. Another reason for this project being more successful is that I was able to take lessons learned from my bachelors project, such as involving my supervisor more and using more resources available to me.

If I was to redo this project, I would focus more on understanding different parallel frameworks before beginning development, and how different versions of these libraries affect their performance. Another thing that I would redo is to be more rigorous in following test driven development, as I have developed a parallel function before the tests. This had a negligible effect on the development of the software, but does encourage poor habits when developing code.