Scaling a hippocampus model with GPU parallelisation and test-driven refactoring

Jack Stevenson and Charles Fox

School of Computer Science, University of Lincoln, UK

Abstract. The hippocampus is the brain area used for localisation, mapping and episodic memory. Humans and animals can outperform robotic systems in these tasks, so functional models of hippocampus may be useful to improve robotic navigation, such as for self-driving cars. Previous work developed a biologically plausible model of hippocampus based on Unitary Coherent Particle Filter (UCPF) and Temporal Restricted Boltzmann Machine, which was able to learn to navigate around small test environments. However it was implemented in serial software, which becomes very slow as the environments and numbers of neurons scale up. Modern GPUs can parallelize execution of neural networks. The present Neural Software Engineering study develops a GPU accelerated version of the UCPF hippocampus software, using the formal Software Engineering techniques of profiling, optimisation and test-driven refactoring. Results show that the model can greatly benefit from parallel execution, which may enable it to scale from toy environments and applications to real-world ones such as self-driving car navigation. The refactored parallel code is released to the community as open source software as part of this publication.

1 Introduction

The hippocampus [2] is an important area of the brain involved in spatial memory. It is known to represent self-location and views of high-level objects, and to compute with them for current, replayed, and predicted times, forming an ego-centric map, planner, and episodic memory. These are tasks also required by mobile robots such as self-driving cars as they localise, map, and plan around their environments [2]. Hippocampal models might thus be used to improve these robots' abilities beyond current Simultaneous Localisation and Mapping (SLAM) systems towards more human levels.

A recent model of the hippocampus is the Unitary Coherent Particle Filter (UCPF) [6], [7]. This model maps the wake-sleep algorithm [9] in a Temporal Restricted Boltzmann machine [12] onto biologically plausible structures and processes of hippocampal areas. The model is notable for predicting the need for after-depolarisation potential (ADP) to be found and used in region CA3 (sub-field 3 of the cornu ammonis), as occurs in the biology, to enable the wake-sleep phases via the theta rhythm. The model also makes use of a cholinergic

Subiculum-Septum (Sub-Sep) pathway to detect and correct for lostness of localisation. The model architecture is reviewed in more detail below.

The UCPF model was then scaled up [11], on a serial CPU, to perform a robotic navigation task shown in fig. 1b. Here a simulated agent random walks around 13 discrete locations, with four possible orientations at each, and each having a pre-computed bag of SURF (Speeded Up Robust Features) visual features taken from a real world mobile robot environment. Simulated odometry is also computed and used to create grid cell activations. Entorhinal Cortex (EC) thus contains visual and grid cell inputs. The UCPF model was able to successfully learn and navigate around this environment in the presence of realistic sensory noise and visual ambiguity, including recovering from loss of localisation using the Sub-Sep system.

However, at the software implementation level, this serial software becomes slow as the environments and numbers of neurons scale up. The plus maze task 1b was limited to 13 locations and 86 CA3 neurons for this reason. Modern GPU architectures [10] can massively parallelize execution of neural networks. This is usually done to accelerate backpropagation of multilayer perceptrons for non-biologically plausible machine learning [3]. But they similarly offer the possibility of speeding up more realistic biological networks such as the UCPF hippocampus.

We here use GPUs, together with the formal software engineering techniques of profiling, optimisation and test-driven refactoring [5], to parallelise the learning process of the UCPF hippocampus software. By parallelising, we enable the easy horizontal scaling of the model by reducing the workload on a given processing unit. The software engineering techniques used to ensure compatibility with the original model and to guide optimisation are intended as a case study for similar accelerations of other neurally plausible models. The newly engineered fast and scalable code is released to the community as open source software as part of this publication, as may be useful to applied robotics researchers in self-driving and similar fields.

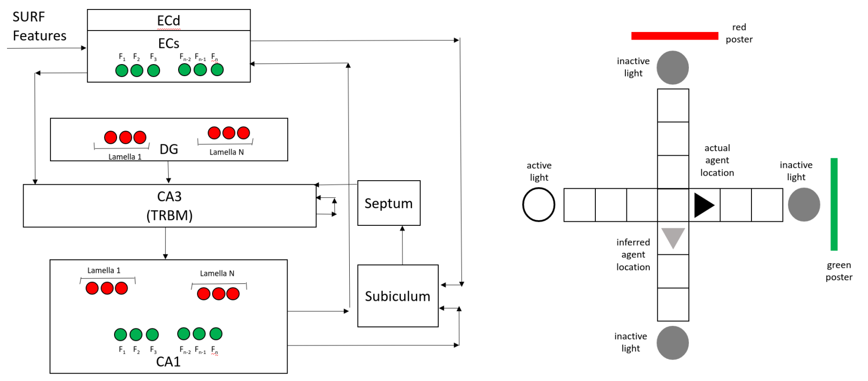
2 Hippocampus review

2.1 Neuroanatomy

The hippocampus is part of hippocampal formation system along with the EC and Dentate Gyrus (DG) [2]. EC may be divided into superficial layers (ECs) and deep layers (ECd) 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 sensory input from external sources via ECs. ECs input can include high level object perception from cortical areas, and an odometric percept encoded in EC grid cells. ECs output is then passed to the DG which sparsifies the EC representation. DG and ECs output are passed to the CA3, which has strong recurrent connections, which are disabled by the presence of Septal acetylcholine

(ACh). CA3 then projects its output to CA1, which projects the apparent output of the system to ECd. ECs and CA1 also project to Subiculum (Sub), which projects to Septum (Sep) to activate ACh.

2.2 Unitary Coherent Particle Filter model



(a) (b)

Fig. 1: (a) Illustration of the hippocampus model showing data flows and hippocampus regions. SURF features are used in this case as the visual inputs, from [11]. (b) The plus maze environment the model is learning, from [6].

UCPF models the areas ECs, DG, CA3, CA1, ECd, Sub and Sep as seen in fig. 1a. Sensor input from any available modality is placed in ECs as input. This can include biologically realistic sensors such as high level object detectors and grid cells, and/or machine vision and robotics sensors such as visual SIFT or lidar features at egocentric locations. As in many hippocampal models [2], DG is assumed to sparsify this representation using PCA encoding. CA1 is assumed to be an output using the same sparsified encoding in the same basis as DG, and ECd a de-sparisfied output encoding in the same basis as ECs.

DG and ECs project to the CA3, which functions in the model as the hidden layer in a modified Temporal Restricted Boltzmann Machine (TRBM) [9], whose hidden variable (neurons) have joint probability distribution:

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

where z represents a Boolean observation vector from DG and ECs, x represents the Boolean hidden state vector of CA3, the prime symbol (') denotes appending an extra dimension containing bias 1 to a vector, and $W_{x'_t x'_t}$ and $W_{x'_t z'_t}$ are the weight matrices connecting these vectors.

The hidden variables function similarly to those in a Hidden Markov model which is most commonly used to fuse sensory input over time with prior memories in order to de-noise the current input and accurately localise from noise inputs. They can also be used to replay memories and predict future plans when disconnected from sensor inputs.

The TRBM is thus a fundamentally serial processing method, with each successive sample of the CA3 state over time requiring the previous sample's result as input. This means that there are limited points of parallelisation in the learning process. However x is a vector state so within each sequential step, the computation can be parallized as the individual CA3 neurons.

UCPF uses a modification of the TRBM, computing a deterministic maximum *a posteriori* (MAP) estimate $\hat{x}_t$ rather than drawing probabilistic samples,

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

This makes CA3 track the MAP states of location and de-noised sensors, in the sparsified basis, which is decoded by CA1 and placed in ECd as output.

Sub/Sep detect loss of tracking by comparing the input and output sensor values (including localization estimates), and sending tonic ACh to CA3 when they differ above a threshold. Thus when tracking is lost, CA3 disables the use of prior information and uses only the current sensory input to relocalise. This can occur both when exploring novel environments and when tracking has been lost by the algorithm in known environments.

The first version of UCPF used hand-set weights throughout, in order to demonstrate the biological plausibility of localisation. The model was then extended [7] to replace these with learning of the weights, in a biologically plausible way. This mapped stages within the 10Hz theta oscillation[2], controlled by phasic ACh, to the inference and learning stages of the wake-sleep algorithm,

$$\Delta w_{ij} = \alpha(\langle x_i pop_j \rangle_{P(x|pop,b)} - \langle x_i pop_j \rangle_{P(pop,x|b)}), \tag{3}$$

where the input population *pop* includes all of the CA3 input populations: ECs, DG and CA3 recurrents, and b are prior biases (detailed in [6]). The UCPF model thus integrates inference and learning, which are performed in alternating states of the the theta cycle. There are no distinct 'offline training' and 'on-line inference' runs of the model – the two steps always run together, as postulated in the real, awake hippocampus.[1] Software optimisation should thus target both learning and inferring together.

[1] The 'wake-sleep' algorithm is a machine learning structure which is here unrelated to night time sleep behaviour of the hippocampus. Night time slow wave sleep is thought to be involved in consolidating memories from hippocampus to cortex so is outside the scope of the UCPF model.

3 Experiment design

We would like to locate each of the bottlenecks in the serial implementation and then optimise them with TensorFlow parallelizations. Our first experiment thus consists of an iterated process of profiling and parallalization refactoring in response to each bottleneck found. It is of general interest to report the sequence of bottlenecks found as they may be common to other biologically realistic neural models and give some insight into what optimisations could be useful in this class of models. Many algorithms have some inherently serial component in addition to parallelizable components, so when the bottleneck becomes an inherently serial component we stop doing refactorings.

Second, we would then like to know and report what total speedup is created once all parallelizable optimisations have been made, and how it varies as a function of the size of the hippocampus simulated, i.e. the number of neurons in CA3. While currently available GPUs are limited in size, trends observed here may suggest how real-time simulation will be able to grow as GPUs become larger.

4 Methods

4.1 Task configuration

For all experiments, we run the UCPF model in the same configuration. The plus maze environment of fig. 1b was simulated, and a random walk paths for the agent within this environment simulated for 3000 steps per epoch over 10 epochs, as in [11]. Times were recorded using the Python *time* function on an i5-8700@3.20GHz, 24GB DDR4 RAM@2600Mhz and an 11GB NVidia Geforce 1080TI. Open source code is available from `github.com/A-Yakkus/hclearn/`.

4.2 TensorFlow

For each refactoring step, we rewrite the top bottleneck model using TensorFlow [1], a data flow library that performs low level math operations in parallel on GPU. This level is suitable for mimicking biological models in software, at the level of modelling particular biological characteristics of individual neurons. In TensorFlow is it possible to define a function to represent a neurons's mapping from its inputs to its output, then make many parallel copies of this function to make a population of neurons and deploy them on the GPU. This method is used here.

4.3 Formal refactoring process

At each refactoring step, we wish to speed up the top bottleneck whilst keeping the code's behaviour the same. A well known challenge of refactoring is ensuring that each modified version retains the same external behaviour as its predecessors. To ensure this, we apply Fowler's formal refactoring methodology [5]. This consists of three stages for each refactoring:

First we profile the code, this tells us which functions are using the most amount of time to run, in terms of the total time spent running the function and the cumulative time, which includes the run time of external functions. This provides the list of program bottlenecks to refactor, which we can rank based on the total time spent in the function.

Second, we write a suite of unit tests around the top bottleneck of the code. This is done by running the original units with various inputs, observing their outputs, and writing tests which assert these outputs follow from these inputs.

Third, we refactor the code in the bottleneck unit using TensorFlow, and rerun the refactored version through the unit tests. These identify any bugs in the new version which can be fixed. A refactoring is only considered complete when it passes the unit tests.

5 Results

We present two types of result from the above experiments. First, we report on what bottlenecks were found and how each was optimised away. These results may be of interest to authors of other neural models as they may suggest similar bottlenecks in them.

Second, we test the performance of the final optimised system as a function of CA3 size. This shows how the speedups scale with larger models, and suggests how they might continue to scale with larger future GPUs beyond what is currently available.

5.1 Bottleneck locations

The profiling results for the original, unoptimised *serial* code, are shown in fig. 2a. These show that there were three main bottlenecks, each taking similar total time: computation of the Boltzmann probabilities functions; the Kronecker outer product between vectors; and Bayesian fusion of probabilities.

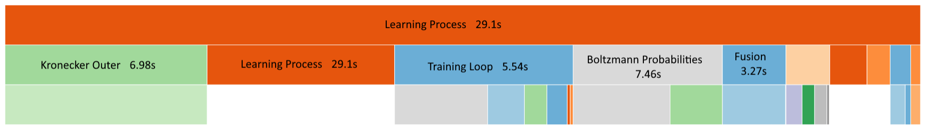
Computation of Boltzmann probabilities is used to calculate the probability P_{on} of a single neuron being on given its input vector x and weight vector w, used in equation 1. The exp(0) term is the probability of the neuron being off, equal to $\exp(-(\mathbf{0}.\mathbf{w})) = 1$, and is here used to compute the normalizing factor Z in that equation,

$$P_{on} = \frac{\exp(-(\mathbf{x}.\mathbf{w}))}{\exp(-(\mathbf{x}.\mathbf{w})) + \exp(0)} \tag{4}$$

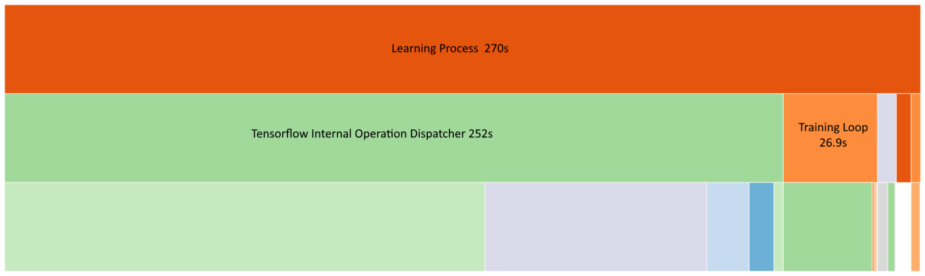
The Kronecker outer product $\mathbf{x} \otimes \mathbf{pop}$ is used in the computation of eqn. 3, for the expectations in the wake and sleep terms.

Bayesian fusion is used both in inference (eqn. 1) to fuse probabilities computed for sensory and recurrent connections, and also in learning (eqn. 3) to fuse probabilities arising from observations and biases,

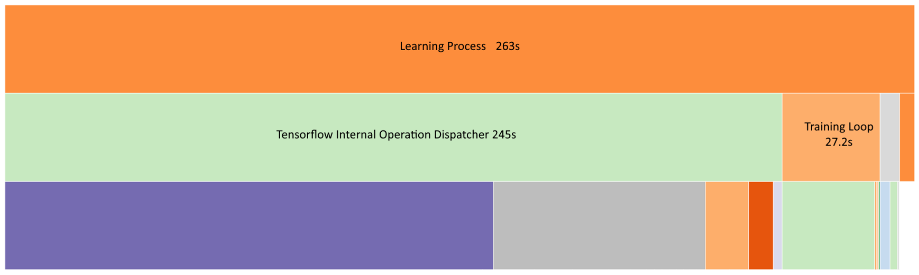
$$f(p_{x1}, p_{x2}) = \frac{p_{x1}p_{x2}}{(p_{x1}p_{x2}) + (1 - p_{x1})(1 - p_{x2})} \tag{5}$$



(a)



(b)



(c)

Fig. 2: Profile diagrams at each stage of the refactoring process. Each row is one level of the program call stack, with the top row being the lowest point. Each bar represents the percentage of time the named function took of the above row, with the times provided being the cumulative time of each function in the entries execution. (a) Initial profile of the learning process. (b) Profiling after the first round of optimisations as described by equations 4 - 6 (c) Final profiling of the system. This shows that most of the time is spent in passing data between the CPU and GPU.

An additional bottleneck occurred in computing the sigmoid function for neuron activations,

$$f(x) = -\log(\frac{1}{x} - 1) \tag{6}$$

We refactored each of these functions using parallel TensorFlow. The Boltzmann probabilities and Bayesian fusion are easily parallelized: equations 4 and 5 are functions performed by single, independent neurons. These were thus coded in TensorFlow then deployed on the GPU as many parallel copies for all the neurons in each relevant population. TensorFlow does not have a native implementation of the Kronecker outer product, so we created our own. The sigmoid functions were changed to be computed for all neurons simultaneously on the GPU rather than in series on CPU.[2] Profiling results after these optimisations can be seen in fig. 2b.

Interestingly, after refactoring, the time to execute on the original model first *increased* by 9x on the original task. This effect comes from the Tensorflow operation dispatcher, which handles the compilation of the neural graph and transfers data between the system memory and GPU Memory. Thus at smaller scales, the refactoring process appears to be detrimental to system performance. However as we will discuss in section 5.2, the move to Tensorflow shows benefits as we scale up.

Casting. At this point in the refactoring process, the new bottleneck can be seen to be in casting between the datatypes used in the CPU and GPU as data is transferred between them.

This is an issue which had arisen from the introduction of our own new TensorFlow refactorings rather than from the original code. The original system loaded weights in CPU as 64-bit floats while TensorFlow initialises variables as 32-bit floats. Bottleneck time was spend converting between them. This lead to an investigation in how to use a Boolean tensor as a tensor of integers of zeros and ones for math operations. TensorFlow provides two ways to handle this: type casting and the *where* function. Type casting creates a new tensor of the requested data type and populates it with the original values made to fit into the requested data type. On the other hand, the where function takes a condition, and returns one of two values based on if the condition is true or false. Table 1 shows that there is an approximate 33% time reduction when casting to a given data type to using the *where* function.

Number of Nodes	Time for tf.cast (s)	Time for tf.where (s)
10	62.073	43.839
100	62.091	43.934
1000	62.208	44.035
10000	61.614	44.288

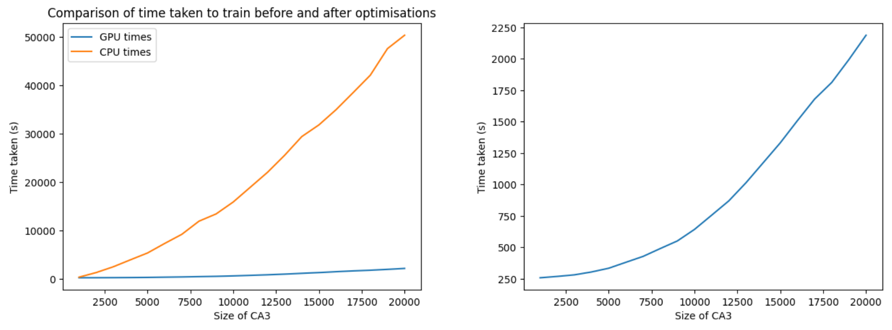
Table 1: Timing 100000 invocations of each function over different numbers of nodes. The times reported are cumulative of all 100000 invocations.

[2] Sigmoid functions were found to be a bottleneck in pure machine learning DNNs, where they are now replaced by rectified linear units (ReLUs) for speed. However sigmoids are required in UCPF for biological plausibility, and our aim in refactoring is to preserve neural functionality rather than make such approximations.

Profiling results after removing unnecessary casting operations can be seen in fig. 2c. The bottleneck is now in transferring data between inherently serial CPU and parallel GPU functions which is not easily further optimised, so we stop refactoring here.

5.2 Final system performance

Fig. 3a shows the total time to run the model as a function of CA3 size on CPU and GPU. Fig. 3b shows the same GPU time curve zoomed in on the y-axis to better show its shape. As the size of CA3, measured in number of neurons, is increased, the GPU implementation's advantage over the original CPU implementation increases, reaching a 23x speedup at the maximum 20,000 neurons size CA3 tested. This maximum is the limit of neurons possible to physically instantiate in parallel on the available GPU hardware.



(a) (b)

Fig. 3: Line graphs showing the time taken to train the model with extra nodes in steps of 1000 from 1000 to 20000. (a) Comparison between CPU and GPU models. (b) The same GPU times as in the comparison, shown alone with a zoomed y-axis.

6 Discussion

If larger GPU's were available, we would expect the trends in fig. 3a and 3b to continue to larger CA3s. When comparing the two lines in fig. 3a, the growth of the GPU line appears to linear, but on further inspection, the growth follows a more quadratic curve (fig. 3b). This indicates that whilst using a GPU does provide significant benefits over the CPU, there are still further optimisations within the training process that can be made. This may be because the UCPF model assumes that all N CA3 cells are mutually connected by N^2 recurrent connections. While FPGA style parallelism might compute these N^2 signals in $\mathcal{O}(1)$ time using N^2 physical wires, GPUs do not immediately connect all their processing elements in this way, and require additional time to move information around hierarchies of elements. The model could be easily modified to be more like the biological hippocampus, which is thought to have many but not fully connected recurrent connections, which could reduce this complexity. (A similar idea has recently worked well to speed up machine learning DNNs as attention and transformers.)

The original implementation was designed to run with a CA3 size of 86 neurons, so scaling to 20,000 is a large improvement. For comparison, the real biological hippocampus in Wistar rats has around 320,000 neurons [4], and in humans around 2 million [8]. The current GPU used cannot fit these biological scales of neurons into memory, but is now only one power of ten away from the rat, and two powers of ten from the human.

The aim of this study is only to show speed gains and the ability to scale up the number of simulated neurons – not to test for the localisation accuracy of the larger CA3 systems enabled by the software. We used the same constrained plus maze task as the original implementation in all tests. This task was designed to be solvable by a small number (86) of CA3 neurons. It is likely that the larger CA3s enabled by the new implementation, perhaps in conjunction with larger GPUs or other neural hardware accelerators such as FGPAs and ASICs will allow future work to try mapping and localising in larger environments, such as those of self-driving cars.

References

1. Abadi, M.: Tensorflow: learning functions at scale. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 1–1 (2016)
2. Andersen, P., Morris, R., Amaral, D., Bliss, T., O'Keefe, J.: The hippocampus book. Oxford university press (2006)
3. Bengio, Y., Goodfellow, I., Courville, A.: Deep learning, vol. 1. MIT press Cambridge, MA, USA (2017)
4. Boss, B.D., Turlejski, K., Stanfield, B.B., Cowan, W.M.: On the numbers of neurons on fields ca1 and ca3 of the hippocampus of sprague-dawley and wistar rats. Brain research **406**(1-2), 280–287 (1987)
5. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
6. Fox, C., Prescott, T.: Hippocampus as unitary coherent particle filter. In: Neural Networks (IJCNN), The 2010 International Joint Conference on. pp. 1–8 (2010)
7. Fox, C., Prescott, T.: Learning in a unitary coherent hippocampus. In: Artificial Neural Networks (ICANN) (2010)
8. Harding, A., Halliday, G., Kril, J.: Variation in hippocampal neuron number with age and brain volume. Cerebral cortex (New York, NY: 1991) **8**(8), 710–718 (1998)
9. Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural computation **18**((7)), 1527–1554 (2006)
10. Owens, J.: GPU architecture overview. In: ACM SIGGRAPH 2007 courses, pp. 2–es (2007)

11. Saul, A., Prescott, T., Fox, C.: 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 (2011)
12. Sutskever, I., Hinton, G.E., Taylor, G.W.: The recurrent temporal restricted boltzmann machine. Advances in neural information processing systems **21** (2008)