A General-Purpose GPU Reservoir Computer

A thesis submitted in partial fulfilment

of the requirements for the Degree of

Master of Electrical and Computer Engineering

in the University of Canterbury by

Tūreiti Keith

University of Canterbury

Christchurch, New Zealand

1st April 2013

Supervisor

Dr. Stephen J. Weddell

Associate Supervisor

Dr. Andrew Bainbridge-Smith

Abstract

The reservoir computer comprises a reservoir of possibly non-linear, possibly chaotic dynamics. By perturbing and taking outputs from this reservoir, its dynamics may be harnessed to compute complex problems at "the edge of chaos". One of the first forms of reservoir computer, the Echo State Network (ESN), is a form of artificial neural network that builds its reservoir from a large and sparsely connected recurrent neural network (RNN). The ESN was initially introduced as an innovative solution to train RNNs which, up until that point, was a notoriously difficult task. The innovation of the ESN is that, rather than train the RNN weights, only the output is trained. If this output is assumed to be linear, then linear regression may be used.

This work presents an effort to implement the Echo State Network, and an offline linear regression training method based on Tikhonov regularisation. This implementation targeted the general purpose graphics processing unit (GPU or GPGPU). The behaviour of the implementation was examined by comparing it with a central processing unit (CPU) implementation, and by assessing its performance against several studied learning problems. These assessments were performed using *all 4* cores of the Intel i7-980 CPU and an Nvidia GTX480. When compared with a CPU implementation, the GPU ESN implementation demonstrated a speed-up starting from a reservoir size of between 512 and 1,024. A maximum speed-up of approximately 6 was observed at the largest reservoir size tested (2,048). The Tikhonov

regularisation (TR) implementation was also compared with a CPU implementation. Unlike the ESN execution, the GPU TR implementation was largely slower than the CPU implementation. Speed-ups were observed at the largest reservoir and state history sizes, the largest of which was 2.6813. The learning behaviour of the GPU ESN was tested on three problems, a sinusoid, a Mackey-Glass time-series, and a multiple superimposed oscillator (MSO). The normalised root-mean squared errors of the predictors were compared. The best observed sinusoid predictor outperformed the best MSO predictor by 4 orders of magnitude. In turn, the best observed MSO predictor outperformed the best Mackey-Glass predictor by 2 orders of magnitude.

Acknowledgements

Thanks to Dr. Stephen Weddell for his excellent supervision and guidance throughout this process. Thanks also to Ms. Lisa Carter, Dr. Philippa Martin, Dr. Andrew Bainbridge-Smith and the staff at the Dept. of Electrical and Computer Engineering, University of Canterbury for supporting my application to write this thesis from abroad. Thanks to Mr. David van Leeuwen for maintaining the test platform and repositories used in this work. And of course, a special heartfelt thanks to my mainstay, my wife, Yana, for her unconditional love and support.

Contents

1 Introduction

First described in a neuro-anatomical context as the *temporal recurrent network* [1, 2], the first computational *reservoir computer* (*RC*) models were introduced as the *Echo State Network* (*ESN*) [3], and the *liquid state machine* [4]. These reservoir computers comprise a dynamical *reservoir*. Inputs may perturb this reservoir, which maps to a higher dimensional space for computation. Outputs may be tapped from the reservoir, thus mapping the reservoir state to a typically lower-dimensional output. Thus, the possibly non-linear, possibly chaotic behaviour of the reservoir may be harnessed [5] to compute complex problems "at the edge of chaos" [6].

The structure of the reservoir computer is supported by a proposed unification of several classical recurrent neural network gradient descent methods, an RC predecessor, referred to as Atiya–Parlos Recurrent Learning (APRL) [7, 2]. RC is further validated by the Back-Propagation Decorrelation method; a simplification of the APRL method that adopts a network structure, similar to, but less restrictive than the ESN, the focus of this work [8, 2].

The Echo State Network is a form of *artificial neural network* (*ANN*) [9]. These networks are inspired by the construction of the brain, and are typically constructed through a *learning* or *training* process. ANNs are composed of computational nodes called *neurons* which are connected via weights. The number of neurons and the weights connecting the neurons determines the ANN's behaviour. When training an

ANN, it is typically these weights that are adjusted.

The Echo State Network's reservoir is built from a specific class of ANN called a *recurrent neural network* (*RNN*). The ESN's RNN is typically large and sparsely connected, with hundreds [10] or thousands [5] of neurons. RNNs are notoriously difficult to train, requiring methods such as backpropagation-through-time [11, 12]. The Echo State Network offers an alternative approach to this. Rather than training the individual weights in the RNN, only the output weights are trained. Training may be performed *offline*, before the network goes into service, or *online*, while the network is in service. In offline training, assuming the ESN outputs are linear, the training problem becomes a case of linear regression [3]. This is the innovation of the Echo State Network.

In this work, a configurable Echo State Network and a training method were implemented for the general purpose graphics processing unit (GPGPU or GPU). This follows on from previous work done to implement reservoir computers on hardware. The ESN implemented is that proposed in the original work, [3, 13]. The training method implemented is a form of linear regression called Tikhonov regularisation [14, 15, 2].

Small, portable and high-speed RC applications can be implemented on dedicated platforms such as the Application Specific Integrated Circuit (ASIC) [16] or the Field Programmable Gate Array (FPGA) [17, 18]. These platforms provide platform-mobility at the cost of complexity and solution portability. In contrast, large high-speed networks benefit from a compute-intensive platform such as the Graphics Processing Unit (GPU) [19, 20, 21, 22]. These provide common numerical operations [23, 24, 25] and solution portability, at the cost of platform-mobility.

The work presented in this thesis continues in this vein. The thesis describes the implementation of a configurable Echo State Network and the offline Tikhonov reg-

ularisation training method for the GPU. This work compares the behaviour of this GPU implementation with a CPU implementation. The goals of this work are to ascertain when a GPU ESN implementation can deliver better speed performance than a CPU ESN implementation. More specifically, for what sizes of ESN and for what amount of training data is the GPU better suited. Further to this, a study of the GPU ESN's behaviour in several multi-time-step prediction problems is performed. Three prediction problems are presented to ESNs of various sizes and configurations. The ESN must predict the value of an incoming time-series multiple samples into the future. The problems – a sinusoidal, a Mackey-Glass, and a multiple superimposed oscillators (MSO) time-series – are of varying difficulty. The goal of this work was to compare various configurations of ESN on these problems and to observe the ESN's predictive capability.

This work begins with Chapter 2, which presents the Echo State Network, the Tikhonov regularisation training method, the GPU, and describes the numerical operations required for this implementation. The final sections of Chapter 2 detail the metrics used to perform the speed and predictive performance comparisons presented later in the thesis. Chapter 3 describes the implementation of the ESN and Tikhonov regularisation, the structure of the libraries, and a brief example of how to use them. Chapter 4 gives the results of several experiments performed to assess the behaviour of the GPU ESN implementation on the Intel i7-980 central processing unit (CPU) and an Nvidia GTX480 GPU. Finally, a discussion of the results is given in Chapter 5, alongside ideas for improving and extending this work.

2 Background

Presented in this chapter are the key ideas used in this implementation of a general purpose GPU reservoir computer. The chapter begins with a description of the artificial neural network, and follows with a background on the graphics processing unit. Next, the numerical operations required to implement an ESN are given, and lastly, the metrics used to assess the GPU ESN's behaviour are defined.

2.1 The Artificial Neural Network

The artificial neural network (ANN) [9] is inspired by the construction of the brain. The ANN is used to perform tasks such as function generation, prediction, and classification. Like the brain, an ANN comprises computational nodes called *neurons*, that have potentially many input and output connections to each other. In an ANN, these connections are weighted, and the nodes themselves can be biased. The behaviour of an ANN is determined by several factors including

1. the number of neurons;
2. the activation function of each neuron;
3. the bias applied at each neuron;
4. the topology of the network formed by the neurons; and

5. the weights connecting the neurons, inputs, outputs, feedback, and feed-forward paths.

For a desired network behaviour, these factors are not typically found analytically; rather, they are *learned*.

Artificial neural networks can be loosely broken into two major categories, *feed-forward* and *recurrent* networks. In the feed-forward case, signals move in only one direction through the network, from input to output. There may be one or more layers of neurons connecting the input to the output. When there is only one layer, all neurons are connected by some weight (which may be zero) to the inputs, and also to the outputs. The state of a feed-forward network with a single layer at time n has no dependence on its state at time $n-1$. In a multiple layer feed-forward network, the state of the i^{th} layer at time n depends only on the state of the $(i-1)^{th}$ layer at time $n-1$.

The recurrent network case differs from the feed-forward case, in that the recurrent network includes feedback. This feedback forms a connection between the state of the network at time n and its state at time $n-1$. Further to this, depending on the weights within this network, it is possible for the state of the network at time n to depend on the networks initial state. [26, 27]

2.1.1 Learning in an Artificial Neural Network

In the field of artificial intelligence, learning problems are thought of as either *inductive*, or *deductive*. In deductive (also called *analytical*) learning, the learning agent uses the data it receives, and a set of general rules to form new rules. The new rules must logically entail the more general rules. In inductive learning, the agent uses example data to form a general rule. In this sense, artificial neural networks are

inductive learners. [27]

A general learning agent may use *unsupervised*, *reinforcement*, *supervised*, or *semi-supervised* learning. In the unsupervised case, the agent discovers patterns in the input data. An example of unsupervised learning is *clustering,* where the agent learns to cluster input data into groups. In reinforcement learning, the agent is rewarded or punished based on its performance. Supervised learning requires a set of input and output data pairs from which the agent can learn. Finally, semi-supervised learning is used in problems where there is input data for training, but only some of the input data has corresponding output data. The artificial neural network studied in this thesis is a supervised learner. It is trained using example input and output vectors. [27]

An artificial neural network is capable of either *online* or *offline* learning. In the offline case, the ANN designer assumes that *all* the data processed by the ANN is independent and identically distributed (IID). In other words, that training shall be performed once, and that the data received after training will have some relationship that is either fully or sufficiently captured in the training data. In the online case, the ANN designer assumes that the data processed by the ANN is either not sufficiently IID, or can only be described as IID for discrete periods of its history. Thus, the ANN must continually learn to account for changes to the data over time. [27]

Learning or *training* artificial neural networks can be a complex task, recurrent neural networks especially so. The echo state approach to learning recurrent neural networks was an innovative approach that improved on well known methods such as back-propagation through time [11, 12], and the more recent Atiya-Parlos method [7]. The benefit of the echo state network is that it is relatively easy to train offline. Rather than training the recurrent neural network itself, a set of linear output neurons are trained instead. The following sections describe the Echo State

Network in more detail, and the off-line training method that was used in this work.

2.2 Reservoir Computing

Reservoir computing began in the early 2000's with the work of Herbert Jaeger and Wolfgang Maass [2]. Jaeger introduced the Echo State Network (ESN) in 2001 [3], and Maass first described the Liquid State Machine (LSM) in 2002 [4]. A Reservoir Computer (RC) is a randomly generated dynamic system – a dynamical reservoir with outputs, optional inputs, and optional feedback. The outputs of an RC are formed by tapping signals from the reservoir, and combining them linearly. An RC may also have inputs and feedback that perturb the reservoir. [3, 4, 2]

2.2.1 The Echo State Network

The Echo State Network (ESN) [3] is a reservoir computer based on an artificial neural network. The introduction of the Echo State Network brought about a new paradigm in the learning or *training* of artificial networks. The "echo state approach" decreased significantly the effort required to construct and train a subset of ANNs called recurrent neural networks (RNN). The remainder of this section will describe the Echo State Network and how it is trained.

2.2.2 The Structure of the ESN

Figure 2.1 describes the structure of an Echo State Network. It comprises three parts. The first is a set of K linear *input* neurons, and the third is a set of L linear *output* neurons. The second or central component contains N sigmoidal neurons,

where N is typically large. These central neurons are typically sparsely and randomly connected. The central part of this network is called the *reservoir,* as due to its structure, it can be seen as a dynamical reservoir. The reservoir is dynamical, as its state at time $n + k$ is dependent on its state at time n, and the relationship between these two states can be described using a set of relatively simple equations. [3]

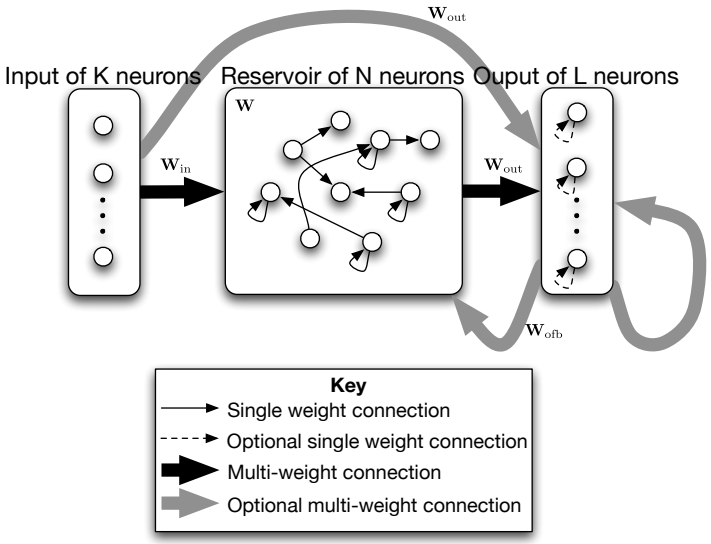

Figure 2.1: The basic architecture of an Echo State Network. Shown is a network of K input neurons, N reservoir neurons, and L output neurons. The reservoir weights $\mathbf{W}$, input weights $\mathbf{W}^{\text{in}}$, output weights $\mathbf{W}^{\text{out}}$, and optional output feedback weights $\mathbf{W}^{\text{ofb}}$ are also shown. Additional unlabelled recurrent output connections can be seen. [3, 22]

The Echo State Network can also be described using two equations. The first,

$$\mathbf{x}(n) = f\left(\mathbf{W}^{\text{in}}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1) + \mathbf{W}^{\text{ofb}}\mathbf{y}(n-1)\right), \tag{2.1}$$

produces an N element vector, $\mathbf{x}$, that describes the *state of the reservoir* neurons at time n. In other words, the current outputs of each of the neurons in the reservoir [3].

Equation 2.1 is a function of a K element *input* vector, $\mathbf{u}$, the state vector from the previous time-step, $n-1$, and the L element *output* vector, $\mathbf{y}$, from the previous time-step. The input, previous reservoir state, and previous output vectors are connected to the current reservoir state through several *weight matrices.* The weights in the $K \times N$ matrix $\mathbf{W}^{\text{in}}$ connect the input vector to the current reservoir state. The weights in the $N \times N$ matrix $\mathbf{W}$ connect the previous reservoir state to the current reservoir state The weights in the $L \times N$ matrix $\mathbf{W}^{\text{ofb}}$ connect the previous output to the current reservoir state. The function $f(\mathbf{.})$ describes the *activation* function of the neurons in the reservoir. This function typically applies a sigmoidal function to an input vector in an element-wise manner, thus producing an output vector of the same size. The sigmoid could, for example, be the hyperbolic tangent function. Thus, for a vector $\mathbf{v}$ with V elements, the activation function would be [3, 13]

$$f(\mathbf{v}) = \begin{bmatrix} tanh(v_1) \\ \vdots \\ tanh(v_V) \end{bmatrix}.$$

The second equation,

$$\mathbf{y}(n) = f_{\text{out}}\left(\mathbf{W}^{\text{out}} \begin{bmatrix} \mathbf{u}(n) \\ \mathbf{x}(n) \end{bmatrix}\right), \tag{2.2}$$

describes the L element output, $\mathbf{y}$, of the network at time n [3]. Equation 2.2 is a function of the current input vector, $\mathbf{u}$, and the current reservoir state, $\mathbf{x}$. The $(K+N) \times L$ matrix weight matrix $\mathbf{W}^{\text{out}}$ describes both the feed-forward connections from the input to the output, and the connections between the current reservoir state and the output. The activation function $f_{\text{out}}(\cdot)$ is typically an identity function. [3]

The Equations 2.1 and 2.2 can also be expressed on a per neuron basis. This expression can perhaps better describe the connection between each neuron and each weight value. Equation 2.1 captures the output of the j^{th} of N reservoir neurons at time n as per

$$x_j(n) = f\left(\sum_{i=1}^{N}\left(w_{i,j}x_i(n-1)\right) + \sum_{i=1}^{K}\left(w_{i,j}^{\text{in}}u_i(n)\right) + \sum_{i=1}^{L}\left(w_{i,j}^{\text{ofb}}y_i(n-1)\right)\right).$$

Here, the i^{th} reservoir neuron output from the previous time-step is weighted by $w_{i,j}$. The i^{th} input is weighted by $w_{i,j}^{\text{in}}$, and the i^{th} output from the previous time-step is weighted by $w_{i,j}^{\text{ofb}}$. [3, 26]

Similarly, Equation 2.2 can be described as

$$y_j(n) = f_{\text{out}}\left(\sum_{i=1}^{K+N} w_{i,j}^{\text{out}}v_i(n)\right).$$

Which gives the j^{th} of L outputs at time n. Here, the value v_i is weighted by $w_{i,j}^{\text{out}}$. The value $v_i(n)$ is the i^{th} value of the vector $\mathbf{v}$ at time n, where [3, 26]

$$\mathbf{v}(n) = \begin{bmatrix} \mathbf{u}(n) \\ \mathbf{x}(n) \end{bmatrix}.$$

2.2.3 Constructing an ESN

When constructing an Echo State Network, the weights in matrices $\mathbf{W}^{\text{in}}$, $\mathbf{W}$, and $\mathbf{W}^{\text{ofb}}$ (Equation 2.1) must be generated; only the matrix $\mathbf{W}^{\text{out}}$ is learned. Previous work looks into the structure of these matrices, especially the reservoir weight matrix $\mathbf{W}$. This is discussed in more detail in Section 2.2.4. For this work, ESNs were constructed using the method presented by Jaeger in [13]. This involves several heuristics or "rules-of-thumb" collated through his experience.

The weight matrix, $\mathbf{W}$, represents the connections within the reservoir at the core of the Echo State Network (Equation 2.1). This is a randomly generated, sparse matrix. To achieve the "echo state" property described in Section 2.2.4, and to maintain stability, the spectral radius of $\mathbf{W}$,

$$\rho\left(\mathbf{W}\right) = max\left(\left|\lambda\left(\mathbf{W}\right)\right|\right), \tag{2.3}$$

should be less than 1 [13]. Here $\lambda\left(\cdot\right)$ returns a vector of eigenvalues of a given matrix, $\left|\cdot\right|$ returns the element-wise absolute value of a vector, and $max\left(\cdot\right)$ returns the maximum value of a vector. One method to produce such a matrix is

$$\mathbf{W} = \frac{\rho \mathbf{W}_{\text{rand}}}{max\left(\left|\lambda\left(\mathbf{W}_{\text{rand}}\right)\right|\right)}. \tag{2.4}$$

Here, a random matrix, $\mathbf{W}_{\text{rand}}$, is divided by its spectral radius, then multiplied by some desired spectral radius, ρ [13]. The non-zero values of the matrix $\mathbf{W}_{\text{rand}}$ typically consist of values drawn from a uniform distribution. In [13], they are drawn from a uniform distribution over the range $[-1, 1]$.

The size, N (see Section 2.2.2) of the reservoir is also an important consideration. In the *offline training* case presented in [13], Jaeger states that the size of the reservoir should be relative to, T, the amount training data available. He gives a rule of thumb for the size of the reservoir as

$$\frac{T}{10} \leq N \leq \frac{T}{2}. \tag{2.5}$$

The amount of training data (T) required is dependent on the problem. Sufficient information must be captured in the training data for the ESN to behave correctly when executing on new data (see Section 2.1.1).

The remaining weight matrices, $\mathbf{W}^{\text{in}}$ and $\mathbf{W}^{\text{ofb}}$ are also typically drawn from a uniform distribution. The range of these values will impact on the linearity of the input and feedback into the system. Larger weight values will scale the input and feedback closer to the saturation range of the reservoir neurons, smaller weight values will scale input and feedback closer to the linear range of the input neurons. [13]

2.2.3.1 A Systems Theory Perspective

Recent work [28] has looked into ESN construction from a systems theory perspective. The authors propose a metric called the "average entropy of echo states" that quantifies the "richness" of a reservoir's dynamics. They studied the distribution and movement of z-plane poles during ESN execution, by calculating and linearising each state. This work shows that the richness of an ESN's dynamics is improved if the poles of the ESN are evenly distributed within the unit circle. The authors propose an ESN design methodology based on the even distribution of poles in the z-plane.

Also given in [28] is an example of the movement of poles in a dynamically rich system given a sinusoidal input. Here, when the input was near zero, the poles were evenly distributed about the unit-circle. As the input amplitude increased, and the *tanh* neurons were driven into saturation, the poles shrunk towards the origin of the z-plane, thus decreasing the effective spectral radius of the system.

2.2.4 Echo States

The name Echo State Network comes from the behaviour of the ESN's reservoir. Inputs into the reservoir at time n are "echoed" in the reservoir at time $n + k$. A reservoir achieves *echo states* when the influence or "echo" of a reservoir state, $\mathbf{x}(n)$,

on later reservoir states, $\mathbf{x}(n+k)$, tends to zero as $k \to \infty$. A detailed definition and proof of echo states is given in [3].

The quality of a reservoir's echo state can be described using its spectral radius, $\rho(\mathbf{W})$ (Equation 2.3). The spectral radius is inversely proportional to the decay of an echo. *Increasing* the spectral radius will *increase* the influence of the reservoir state, $\mathbf{x}(n)$, on later states, $\mathbf{x}(n+k)$. Thus, the echo from $\mathbf{x}(n)$ decays at a *slower* rate. When *decreasing* the spectral radius, the echo from a reservoir state decays at a *faster* rate. The spectral radius of a reservoir therefore indicates the quality of its memory – a *larger* spectral radius means a *longer* memory. [3]

An Echo State Network designer does not, however, have free range over the size of a reservoir's spectral radius. Rather, this should be tuned to remain within the bounds of stability, and to ensure that the echo states are achieved. To this end, the next section presents a heuristic and three bounds that may be used to achieve echo states.

2.2.4.1 Achieving Echo States

A reservoir is *likely* to achieve echo states when the spectral radius of its weight matrix (Equation 2.3) is [3]

$$\rho(\mathbf{W}) < 1. \tag{2.6}$$

In other words, the matrix $\mathbf{W}$ is *contractive*. This is a good heuristic, but is neither a necessary nor sufficient condition to guarantee echo states. Rather, it is shown in [3] that if $\rho(\mathbf{W}) > 1$; if $\mathbf{u}(n) = \mathbf{0}$, $\forall n$; and if the reservoir activation function $f(\mathbf{v}) = tanh(\mathbf{v})$; then the reservoir *will not* have echo states. [2, 3]

A *sufficient* condition to achieve echo states is given in [3] as

$$\sigma_{\max}(\mathbf{W}) < 1. \tag{2.7}$$

This states that the largest single value of the matrix $\mathbf{W}$ must be less than 1. This is true for any input $\mathbf{u}$, but is proven only for reservoir activation function of $f(\mathbf{v}) = tanh(\mathbf{v})$.

Subsequent work has shown that the condition given in Equation 2.7 is conservative. Echo states can be achieved when

$$inf_{\mathbf{D}\in\mathcal{D}}\, \sigma_{\max}\left(\mathbf{DWD}^{-1}\right) < 1.$$

Where $\mathbf{D}$ is an arbitrary matrix from the set $\mathcal{D} \subset \mathbb{R}^{N\times N}$, that minimises the "D-norm" of the reservoir weights, $\|\mathbf{W}\|_{\mathbf{D}} = \sigma_{\max}\left(\mathbf{DWD}^{-1}\right)$. The $inf_{\mathbf{D}\in\mathcal{D}}$ function describes the infimum of the subset $\mathcal{D}$. This is the greatest element in $\mathbb{R}^{N\times N}$ that is less-than or equal-to all elements of $\mathcal{D}$. [2, 29]

2.2.5 Training an ESN

Since the introduction of the echo state approach to recurrent neural network training [3], both alternatives and extensions to this reservoir computing approach have been proposed [2]. Alternatives include the online *backpropagation-decorrelation* method [8], and the evolutionary *evolino* method [30]. Extensions include the *Tikhonov regularisation method* [2, 15, 14] for offline training, and the *recursive least squares* method for online training [5]. This thesis looks into the classical ESN training method proposed in [3, 13], and the Tikhonov regularisation method to learning output weights. This section summaries the classical training method and

the Tikhonov method.

2.2.5.1 Offline Training

Given a set of $T+1$ known *training* inputs

$$\mathbf{U}_{\mathrm{t}} = \left[\begin{array}{ccc} \mathbf{u}_{\mathrm{t}}(0) & \ldots & \mathbf{u}_{\mathrm{t}}(T) \end{array} \right],$$

and known *training* outputs,

$$\mathbf{Y}_{\mathrm{t}} = \left[\begin{array}{ccc} \mathbf{y}_{\mathrm{t}}(0) & \ldots & \mathbf{y}_{\mathrm{t}}(T) \end{array} \right], \tag{2.8}$$

a history of T reservoir states can be captured using Equation 2.1. To do this, the known inputs and outputs are substituted into Equation 2.1 as per

$$\mathbf{x}_{\mathrm{t}}(n) = f\left(\mathbf{W}_{\mathrm{in}}\mathbf{u}_{\mathrm{t}}(n) + \mathbf{W}\mathbf{x}_{\mathrm{t}}(n-1) + \mathbf{W}_{\mathrm{ofb}}\mathbf{y}_{\mathrm{t}}(n-1)\right). \tag{2.9}$$

Then, the values of $\mathbf{x}_{\mathrm{t}}(n)$ are calculated for $1 \leq n \leq T$, where $\mathbf{x}_{\mathrm{t}}(0)$ is some initial reservoir state, for example the null vector, $\mathbf{0}$. Thus we have

$$\mathbf{X}_{\mathrm{t}} = \left[\begin{array}{ccc} \mathbf{u}_{\mathrm{t}}(1) & \ldots & \mathbf{u}_{\mathrm{t}}(T) \\ \mathbf{x}_{\mathrm{t}}(1) & \ldots & \mathbf{x}_{\mathrm{t}}(T) \end{array} \right].$$

Note the use of the known output vector, $\mathbf{y}_{\mathrm{t}}$, in Equation 2.9. This is referred to as *teacher forcing*. This forces the state of the reservoir at time n to be calculated as if the output equation (Equation 2.2) has produced the expected output at time $n-1$. [3, 13]

During the initial state calculation steps, transient noise is expected. To reduce the effects of transients on the training process, some portion of the initial time steps,

$1 \leq n \leq T_0$, should be ignored. This gives

$$\mathbf{X} = \begin{bmatrix} \mathbf{u}_{\mathrm{t}}(T_0) & \dots & \mathbf{u}_{\mathrm{t}}(T) \\ \mathbf{x}_{\mathrm{t}}(T_0) & \dots & \mathbf{x}_{\mathrm{t}}(T) \end{bmatrix}. \tag{2.10}$$

Once captured, $\mathbf{X}$ can be substituted into Equation 2.2 as per

$$\begin{bmatrix} \mathbf{y}_{\mathrm{t}}(T_0) & \dots & \mathbf{y}_{\mathrm{t}}(T) \end{bmatrix} = f_{\mathrm{out}}(\mathbf{W}_{\mathrm{out}}\mathbf{X}). \tag{2.11}$$

Assuming $f_{\mathrm{out}}(\cdot)$ is an identity function, this can be reformulated as

$$\mathbf{Y}_{\mathrm{target}} = \mathbf{W}_{\mathrm{out}}\mathbf{X}. \tag{2.12}$$

Thus, to train the ESN offline, one must solve Equation 2.12. [3, 13]

Perhaps the first solution that comes to mind is $\mathbf{W}_{\mathrm{out}} = \mathbf{Y}_{\mathrm{target}}\mathbf{X}^{-1}$. Unfortunately, $\mathbf{X}$ is not usually invertible. This is because the number of columns in $\mathbf{X}$ is generally much greater than the number of rows (see Equation 2.5). As such, the pseudoinverse can be applied [3, 13], yielding

$$\mathbf{W}_{\mathrm{out}} = \mathbf{Y}_{\mathrm{target}}\mathbf{X}^{+}.$$

2.2.5.2 Tikhonov Regularisation

Also known as *Ridge Regression*, *Tikhonov Regularisation* is given in [2] as a "highly recommendable" choice for learning the matrix $\mathbf{W}_{\mathrm{out}}$. This method is presented as one that reduces numerical instability and the magnitudes of elements in the matrix $\mathbf{W}_{\mathrm{out}}$. Thus, this method reduces sensitivity to over-fitting and noise. [2]

With the state history matrix, $\mathbf{X}$ (Equation 2.10), in hand, Tikhonov Regularisation

can be used to solve Equation 2.12. The solution is given by

$$\mathbf{W}_{\text{out}} = \mathbf{Y}_{\text{target}}\mathbf{X}^T \left(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I}\right)^{-1}, \tag{2.13}$$

where λ is the Tikhonov regularisation parameter. [2, 14, 15]

2.2.5.3 Selecting the Regularisation Parameter

For the problem $\mathbf{A}^*\mathbf{A}\mathbf{x} = \mathbf{A}^*\mathbf{b}$, where $\mathbf{x}$ is unknown and $\mathbf{A}$ is not invertible, a near optimal method for selecting the regularisation factor is given in [31]. Finding the first zero of

$$\hat{g}(\lambda) = \sum_{i=1}^{n} \frac{\beta_i^2 \lambda}{\left(\sigma_i^2 + \lambda\right)^3} - \sum_{i=k}^{n} \frac{\beta_i^2}{\left(\sigma_i^2 + \lambda\right)^2} - s^2 \sum_{i=1}^{k-1} \frac{1}{\left(\sigma_i^2 + \lambda\right)^2},$$

yields a near optimal value of λ. This is generated from the singular value decomposition of the state history matrix, $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^*$. Here, $\beta_i \equiv \mathbf{u}_i^T\mathbf{b}$ (where $\mathbf{u}_i$ is the i^{th} column of $\mathbf{U}$), σ_i is the i^{th} diagonal element of the diagonal matrix $\mathbf{\Sigma}$, and s^2 is the expectation of some i^{th} noise value, $E\left(\epsilon_i^2\right)$.

Unfortunately, this finding can not be applied directly to the problem of ESN training, as the form of the ESN training problem is instead $\mathbf{A}\mathbf{X}\mathbf{X}^\mathbf{T} = \mathbf{B}\mathbf{X}^\mathbf{T}$, where $\mathbf{A}$ is unknown, and $\mathbf{X}$ is not invertible. A similar analytical solution to this problem was not found in the literature.

Another approach to find the Tikhonov regularisation parameter is presented in [15]. Here the authors investigated the relationship between the regularisation factor and the error performance of an ESN. More specifically, they compared ESN mean

squared error performance with the *effective number of parameters* [32],

$$\gamma = \sum_{i=0}^{p} \left(\frac{\sigma_i}{\sigma_i + \lambda} \right). \tag{2.14}$$

Here σ_i is the i^{th} eigenvalue of the $p \times p$ matrix $\mathbf{X}\mathbf{X}^T$. In their approach, an "optimal" regularisation parameter was found by performing a sweep across a range of values. For each regularisation parameter value, the ESN was trained (i.e. $\mathbf{W}_{\text{out}}$ was calculated), and the ESN performance error was measured. To calculate the ESN performance error, the trained ESN was not run over test data; rather, the error was calculated as the mean squared difference between the elements of $\mathbf{Y}_{\text{target}}$ and the resulting $\mathbf{W}_{\text{out}}\mathbf{X}$. Thus, each performance error was calculated in a relatively quick manner. Once the parameter sweep was completed, λ was chosen as the value that yielded the minimum mean squared error.

2.3 The Graphics Processing Unit

The reservoir computer has been applied to non-linear signal processing problems, and has been shown to perform well. Such applications include communications channel equalisation [5], voice recognition [17, 18], and adaptive optics [10]. These applications have motivated research into implementing the RC on either compute-intensive platforms, or small dedicated platforms. Dedicated platforms such as the Application Specific Integrated Circuit [16] or the Field Programmable Gate Array [17, 18] have been used to implement small, portable and high-speed RCs. Compute-intensive platforms such as the Graphics Processing Unit (GPU) have also been used to implement larger ANNs [19, 20, 21, 22].

An early GPU implementation of the ANN [19] demonstrated significant speed-up when matrix operations were migrated to a non-general-purpose GPU. This imple-

mentation took advantage of a knowledge of the GPU architecture, and the graphics application programming interface, to implement matrix multiplication using the rendering hardware on the GPU. A comparison was performed between the ATI RADEON 9700 PRO GPU, and an unnamed CPU. The application was a feed forward ANN image processing application. The GPU purportedly achieved a 20-fold speed-up.

GPU ANN implementations have continued in the same vein. In recent work implementing spiking neural networks on GPGPUs has been shown to achieve significant speed-ups. In a colour image segmentation application, a GPU Spiking Neural Network (SNN) implementation was shown to achieve a significant speed-up. This SNN, when run on an NVIDIA Quadro FX 3800 GPU performed 31 times faster than the Intel(R) Xeon(R) X5550 CPU [20]. Similarly, an exploration of competing GPGPU architectures was performed using the SNN as a benchmarking tool. Two architectures: the *fermi*, represented by Nvidia's Tesla C2050; and the *radeon*, represented by AMD's Radeon 5870, were compared. The fermi was shown to perform better [21]. This survey of the literature did not, however, reveal any previous work in the area of GPU Echo State Network implementations.

Since the late-mid 2000's, programmers have been provided with tools that allow them to use the graphics processing unit (GPU) for general purpose computing. The general purpose GPU (GPGPU) has become synonymous with parallel computing, and graphics devices in general. So much so, that the term GPGPU is interchangeable with GPU, the term for the formerly "non-general-purpose" graphics processing hardware. The benefits of the GPU lie in its capability to process large amounts of data in a highly parallel manner. For a scalable parallelisable problem, the benefits of parallelisation increase with the size of the problem. [33]

Tools available to GPU programmer include OpenCL, an open and cross-platform

toolkit [34]; the heterogeneous computing platform from AMD [35]; and Nvidia's Cuda [36]. For this work, an Nvidia Cuda device was used. This is largely due to the availability of existing Nvidia hardware.

The remainder of this section describes the Nvidia Cuda GPU – its hardware, development toolchain, and programming model. The content of this section comes largely from the *CUDA C Programming Guide,* [37].

2.3.1 The Nvidia Cuda Programming Model

Using Flynn's taxonomy, an NVidia GPU could be described as a *single instruction multiple data* device (*SIMD*). The Nvidia GPU does not, however, behave in entirely an SIMD manner. As such, Nvidia refers to the architecture as *single instruction multiple thread* (*SIMT*). Upon execution, an SIMD device executes the same instruction in parallel on different data. An SIMT device provides both SIMD behaviour and independent thread execution. [37, 38]

The exact capabilities of a Cuda device depend on its *compute capability*, as defined by Nvidia. A device's compute capability is a function of its architecture, and is described by a revision number of the from "$x.y$". Here x is the major revision number determined by the architecture, e.g. either *Tesla* ($x = 1$), *Fermi* ($x = 2$) or *Kepler* ($x = 3$). The minor revision number, y, corresponds to an incremental change to the architecture, and possibly an incremental change to the feature set. [37]

To program an Nvidia Cuda GPU, a programmer writes a *kernel* of code. A kernel can be expressed in a subset of the C^{++} programming language, with some language extensions. These extensions are used to specify kernel execution parameters, and to specify in which part of GPU memory a variable should be placed. Listing 2.1 gives a simple example of a kernel, and Listing 2.2 gives an example of calling a

kernel. The contents of these listings should become clear after reading the following sections. [37]

Listing 2.1 A simple C Cuda kernel that sets all elements in a vector to zero.

```
__device__ /*Execute on a Cuda device*/
void zeroiseVectorKernel(double *vector, int elementCount)
{
    // Calculate the thread index
    int x = (blockIdx.x * blockDim.x) + threadIdx.x;
    // If within the bounds of the vector
    if (x < elementCount)
    {
        // Set this memory address to zero
        vector[x] = 0.0;
    }
}
```

Listing 2.2 A C function that executes the kernel in Listing 2.1. Here, the size of the thread block is set to the size of a warp (see Section 2.3.3.1). In this case, a warp assumed to comprise 32 threads. This value can also be obtained programmatically. The number of thread-blocks (see Section 2.3.1.1) must be calculated to ensure that sufficient threads are called to address the entire vector. The kernel will be queued to execute on the `stream` stream (see Section 2.3.1.2). The block size, grid size and stream are communicated to the kernel via the Cuda-specific `<<<...>>>` syntax. The third argument specifies the amount of shared memory to dynamically allocate per thread-block, this is set to the default value of 0.

```
#define WARP_SIZE 32

__host__ /*Execute on the host*/
void zeroiseVector(
   double *vector,
   int elementCount,
   cudaStream_t stream)
{
   // Use WARP_SIZE threads per thread-block
   dim3 blockDim(WARP_SIZE);
   // The minimum number of thread-blocks required
   dim3 gridDim(ceil(((float)elementCount) / ((float)WARP_SIZE)));
   // Call the vector zeroing kernel.
   zeroiseVectorKernel<<<gridDim,blockDim,0,stream>>>(
      vector,
      elementCount);
}
```

2.3.1.1 The Thread Hierarchy

A single kernel is run across many SIMD threads, where each thread can access *global* memory and *private local* memory. Each running thread is a member of a *thread block*. There may be multiple thread blocks executing at any one time. Each member of a given thread block has access to the same *shared* memory. A thread block is, in-turn, a member of a *grid*. A single grid is associated with the execution of a single kernel. Some Nvidia Cuda devices are capable of executing multiple kernels, and therefore multiple grids in parallel. This memory hierarchy is described

in Figure 2.2. [37]

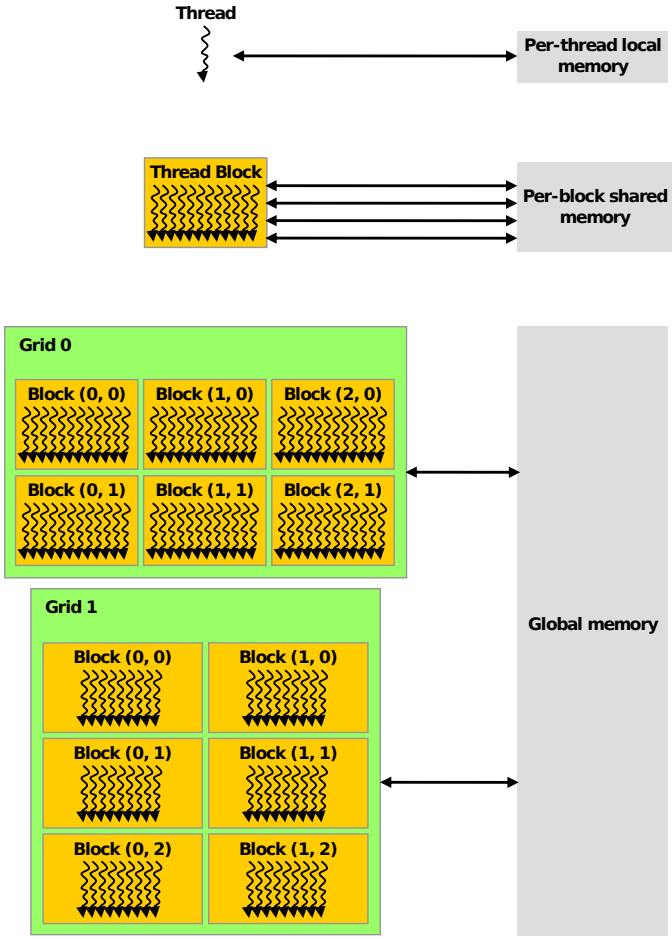

Figure 2.2: GPU memory hierarchy. [37]

An executing kernel runs in parallel across a single grid containing one or more thread blocks. At kernel launch time, the programmer specifies the dimensions of this grid and its thread blocks. Both grids and thread blocks are specified in either one, two, or three dimensions; suiting either vector, plane, or volume problems respectively. Listing 2.2 demonstrates launching a kernel with a one dimensional grid of one dimensional thread blocks. [37]

At execution time, each parallel instance of a kernel has access to information on the thread, block, and grid within which is running. This is called the *thread index*. The programmer can use this information to determine from which data locations to read, and to which to write. Listing 2.2 demonstrates using the thread index to resolve a single value within a vector. [37]

2.3.1.2 Concurrency

The Nvidia Cuda GPU is designed to provide thread level parallelism in an SIMT sense. Cuda devices can also parallelise kernel execution and memory transfer operations. A programmer can take advantage of these capabilities by using *streams*.

A stream can be thought of as a queue of kernel executions and/or memory transfer instructions. A programmer can create a stream and assign kernel executions and memory transfer operations arbitrarily. The operations are executed in the order that they are added. A programmer can achieve kernel-level parallelism by creating multiple streams, and assigning operations to them. Each stream executes its operations sequentially, and attempts to execute operations in parallel with the other streams. Parallelism is not guaranteed, as this depends on both the resources available, and on the capabilities of the GPU. A kernel execution may be parallelised with a transfer from pinned memory to device memory on some devices of compute capability of 1.1 or higher. Two or more kernels may be executed in parallel on some devices of compute capability 2.0 or higher. Listing 2.2 demonstrates launching a kernel with a given stream. [37]

An important aspect of parallel processing is communication between parallel tasks. Suppose the inputs into, for example, kernel a on stream x depend on the outputs of kernel b on stream y. A programmer can specify synchronisation barriers forcing kernel b to wait until kernel a has completed. [37]

2.3.1.3 Memory Considerations

During the execution of an Nvidia Cuda based application, it is usual to transfer data between host memory and GPU device memory. These transfers are typically slower than the accompanying calculations, and can impact heavily on the overall execution time of an application. To reduce this impact, *pinned* or *mapped* memory transfers may be employed, further expert level optimisation is also possible.

Page-Locked & Write-Combining Memory. On systems with a front-side-bus, the programmer can increase the efficiency of host-GPU memory transfers by using host-side page-locked (pinned) and write-combining memory. Pinned memory delivers performance increases when both transferring to and fetching from the GPU memory. Write-combining memory delivers further increases (up to 40 % more), but only where the host writes to this memory. Reads from write-combining memory are relatively slow. Pinned and write-combining memory are generally scarce resources, the programmer should take care not to use them to the detriment of other processes running on the host. [37]

Mapped Memory. In addition to page-locked and write-combining memory, the programmer also has access to mapped memory. A mapped memory allocation operation allocates both page-locked host memory, and GPU device memory. When mapped memory is accessed by a kernel, memory transfers from host to device are performed implicitly and in parallel. A programmer can allocate this memory specifically, or register existing page-locked memory as mapped. [37]

Expert Level Optimisation. A programmer can achieve expert level optimisation by considering specialised memory, and on-device memory configuration.

When executing a kernel on Cuda devices of compute capability 2.0 or higher, the size of the L1 cache and shared memory can be configured. In this case, a multiprocessor uses a single block of physical memory that is partitioned into either L1 cache or shared memory. The programmer can adjust the ratio of shared memory to L1 cache. Listing 2.2 demonstrates specifying the shared memory size with the default value of 0. [37]

A programmer may also consider the impacts of memory coalescing at warp execution time. This concerns specifically the transfer of data from global to shared memory. When a warp requests data from discontiguous addresses scattered around the global memory space, data throughput is slowed. The *NVIDIA CUDA C Programming Guide* [37] contains more information on these topics.

2.3.1.4 Pipelining

A technique referred to as pipelining may also be employed to reduce the impact of memory transfers on execution time. The use of pipelining in computing originates from instruction-pipelining used to increase the instruction throughput of a processor with a single arithmetic logic unit. It works in the same way as a manufacturing production line. There are four stages within the pipeline (or "production-line") fetching an instruction, decoding an instruction, executing the instruction, and writing back the result. Each stage takes the same time to execute and is performed in parallel. When every stage is occupied, an instruction is always ready to be executed. Thus the impact of the fetch, decode, and write-back stages on overall execution time is largely hidden – seen only as the pipeline fills or empties. [39]

This idea can be extended to GPU program design. The impact of data transfers between host and GPU memory can be hidden or reduced by parallelising them with calculations on the GPU. A program may, for example, have three stages –

fetch data from the host, execute operations on this data, return the result to the host. Where fetching data from and returning results to the host are each equivalent or faster than the GPU calculations, a three stage pipeline will hide their impact on overall execution. Where each fetch and return operation is slower than the GPU calculations, their impact will be reduced, but not completely hidden. When designing a Cuda program, streams (see 2.3.1.2) may be used to parallelise data transfers and kernel executions.

2.3.1.5 Double Buffering

Similar to pipelining, *double buffering*, also referred to as *ping-pong buffering*, is a technique employed to parallelise memory transfer operations, and calculation operations. The idea used in ping-pong buffering is to maintain two memory buffers, while one buffer is loaded, calculations are performed on data read from the other buffer. The buffers exchange roles at the next calculation step. [39]

As with pipelining, the idea of double buffering can be extended to the GPU. Buffers can be defined in GPU-Memory (Section 2.3.1.3), and Cuda streams (Section 2.3.1.2) can be used to coordinate the parallel writing to one buffer, and reading from the other.

2.3.2 The Nvidia Cuda Development Toolchain

The Nvidia Cuda toolchain comprises components for compiling and optimising bespoke kernels. It also provides components with existing kernels. While programmers can write, compile, and profile C/C^{++} kernels with the Nvidia Cuda Compiler (`nvcc`), they can also use the existing kernels provided in shared libraries. In version 5.0 of the Cuda toolchain the included libraries are Cuda runtime, CuBLAS, Cur-

and, Cufft, Cusparse, Npp, Thrust, and Cuda math. The details of these libraries are summarised in Table 2.1.

Table 2.1: The Nvidia Cuda toolchain libraries. All floating point operations are available in both single and double precision.

Library	Description
Cuda runtime	Perform device, memory, stream, and event management; error handling, and execution control.
CuBLAS	Perform BLAS level 1, 2, and 3 operations.
Curand	Generate uniform, normal and lognormal pseudorandom sequences.
Cufft	Perform 1, 2, and 3 dimensional Fourier transforms.
Cusparse	Perform Sparse level 1, 2, and 3 operations.
Npp	Signal processing primitives (e.g. filters, colour transforms, and statistical functions).
Thrust	A Cuda implementation of the C^{++} Standard Template Library.
Cuda-math	Perform mathematical operations (e.g. log, sin, tanh, and acos).

2.3.2.1 The Nvidia Cuda Compiler

The Nvidia Cuda compiler (`nvcc`) compiles C/C^{++} code destined to run on either the host, or the GPU device. It is capable of both GPU-device specific compilation, and *just-in-time* compilation.

When writing code, programmers can indicate to the compiler that the code is to execute on either the host or the GPU-device. They can do this using either the `__host__` or the `__device__` qualifiers for host and device code respectively (see the examples in Listings 2.1 and 2.2). When compiling host code, `nvcc` first parses and replaces any Cuda-specific syntax with Cuda runtime functions. This modified code can then be compiled by the host compiler (e.g. `g++`). Device code can be compiled directly to device-specific binaries or to intermediate assembly code. This assembly code, called *parallel thread execution* (or *PTX*) code, can be later compiled

just-in-time upon execution on the target. Compiler arguments are used to specify full or just-in-time compilation behaviour, specifically the `-arch` and `-code`, or the `-gencode` flags. When compiling, the developer can specify the compute capability of the target hardware. In the case of just-in-time compilation, the specified compute capability acts as a minimum bound. Hardware of the specified compute capability or higher will be capable of compiling and executing this code just-in-time. If the target is correctly configured, just-in-time compilation will occur only once, and the resulting binaries will be cached for later use. The cache is invalidated with updates to the Nvidia toolchain.

2.3.3 Nvidia Cuda Hardware

Compared to the CPU, the GPU uses less real-estate for control and caching, and more for parallel processing. This idea is illustrated in Figure 2.3. The Nvidia Cuda GPU comprises a set of multiprocessor devices. At the time of writing, a multiprocessor consists of either 8, 32, 48, or 192 Cuda cores, and a multiprocessor can manage up to 2,048 concurrent threads [37]. These threads are managed at the hardware level, in groups of 32 threads called a *warp*. Within a warp, threads follow a single instruction multiple thread or SIMT behaviour. [37]

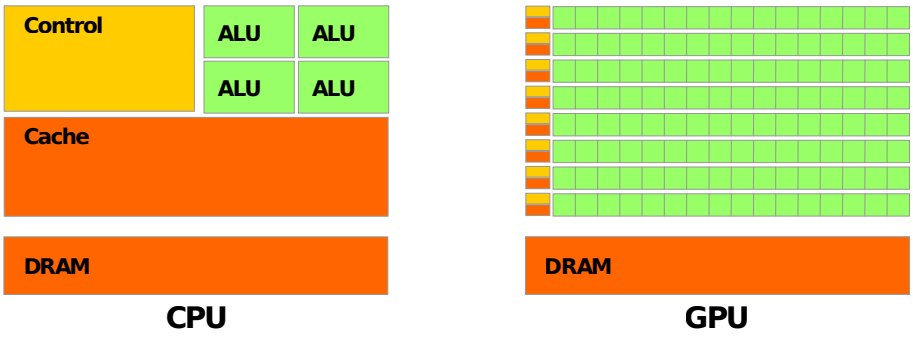

Figure 2.3: A high-level comparison of CPU and GPU architectures. [37]

2.3.3.1 Warp & Weft

The terms warp and weft come from weaving. The warp is a group of parallel threads, the weft is a thread that repeatedly crosses these. Nvidia has borrowed from this terminology. A group of execution threads executing in parallel on the same Cuda core are called a *warp*. Section 2.3.1.1 describes a thread hierarchy in terms of threads, thread blocks, and grids. The actual execution of these threads on the hardware is performed in sets of 32 threads called a warp. At kernel execution time, a multiprocessor is provided with one or more thread-blocks. The multiprocessor breaks these blocks into sets of 32 threads, and spawns each set as a warp. Decisions pertaining to SIMT behaviour are made at the warp level. [37]

2.3.3.2 SIMT Behaviour

As mentioned in Section 2.3.1, Nvidia describes the Cuda GPU as a single instruction multiple thread device. This is a device that can behave both as an SIMD device, and can operate threads independently. The device is most efficient when operating in an SIMD manner. [37]

A kernel strays from SIMD behaviour when it includes a data-dependent conditional branch point. In C/C^{++} this would be an `if` or `if-else` statement, or the conditional tests in a `for` or `while` loop. Kernel branching is assessed at the warp level. [37]

An example of a branch point is given in Listing 2.1, here it is used to ensure that the kernel does not write to addresses beyond the memory allocated to the target vector. Where all threads in a warp agree with the evaluation of this if-statement, SIMD execution shall continue as normal. However, where threads within a warp disagree

with its evaluation, the SIMD execution shall be partitioned. In this example, there are only two possible paths. Each path shall be executed serially, with the agreeing threads executing in parallel, and the other threads temporarily disabled. Full warp-level parallelism resumes once all paths have been evaluated. [37]

In this particular instance, the branching can be avoided at the cost of memory. Suppose that a programmer uses the kernel in Listing 2.1 to set all n elements in a double precision vector $\mathbf{v}$ to zero. The programmer allocates $d \times b \times \lceil n/b \rceil$ bytes of memory to $\mathbf{v}$, where b is the number of threads per thread block and d is the number of bytes allocated to a double precision value. Then the programmer launches the kernel with a block-size of b, and gives the number of elements in the vector as $b \times \lceil n/b \rceil$. In this case, the thread index will never be more than $b \times \lceil n/b \rceil$, and the if-statement will only ever return true. This would, however, leave $d\,(b-1)$ bytes of allocated but unused memory in the worst case.

2.3.3.3 Hardware Multithreading

Associated with each warp are independent program counters and registers. These are maintained on the hardware during the warp's lifetime. This information is maintained in sets of 32-bit registers that reside on the multiprocessor. They are partitioned among the warps. Shared memory, also on the multiprocessor, is partitioned among the thread blocks. The number of available registers and the amount of shared memory is a function of compute capability, and places limits on the number of warps that can reside on a multiprocessor. [37]

2.4 Numerical Operations on the GPU

To implement the Echo State Network on the GPU, several well studied numerical operations, and several bespoke kernels were required. This section outlines these operations.

2.4.1 BLAS Operations

Basic linear algebra subroutines, or BLAS libraries provide vector (level-1), matrix-vector (level-2), and matrix-matrix (level-3) arithmetic operations. [40, 23] Implementations from the Nvidia CuBLAS [23] and the reference BLAS [40] libraries were used extensively in this work. Table 2.2 lists the operations used and the names of the subroutines within which they are implemented.

Table 2.2: Selected BLAS operations.

Description	Operation	Routine Name
The maximum element in a vector, $\mathbf{v}$.	$max\left(\mathbf{v}\right)$	amax
Scale a vector, $\mathbf{v}$ by some scalar value, α.	$\alpha\mathbf{v}$	scal
Add some vector $\mathbf{x}$ to some vector $\mathbf{y}$	$\mathbf{x}+\mathbf{y}$	axpy
Multiply some matrix, $\mathbf{M}$, by some vector, $\mathbf{v}$.	$\mathbf{Mv}$	gemv
Multiply some matrix, $\mathbf{M}_1$, by some matrix $\mathbf{M}_2$.	$\mathbf{M}_1\mathbf{M}_2$	gemm

2.4.2 LAPACK Operations

Linear algebra package, or LAPACK libraries provide more advanced linear algebra subroutines than those found in the BLAS libraries. [41, 42] Implementations from the reference LAPACK [41] and Magma [42] libraries were used in this work. Magma is a hybrid CPU / GPU LAPACK implementation. It implements a selection of

LAPACK routines on the GPU, and interfaces with both the reference LAPACK, and Nvidia CuBLAS libraries to perform LAPACK routines. This work used two operations: singular value decomposition, and eigenvalue calculation. The Magma implementations are discussed here.

2.4.2.1 Singular Value Decomposition

This work uses singular value decomposition (SVD) during Tikhonov regularisation for ESN training. SVD describes techniques for handling matrices that are either singular (non-invertible), or very near singular. SVD can be used where other techniques such as Cholesky, LU, or QR decomposition yield unsatisfactory results. In such cases, SVD can allow one to diagnose the problem, and sometimes to solve it. SVD describes an $M \times N$ matrix $\mathbf{A}$ as:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathbf{T}} \tag{2.15}$$

Where $\mathbf{U}$ is an $M \times N$ column-orthogonal matrix, $\mathbf{\Sigma}$ is an $N \times N$ diagonal matrix with positive or zero elements, and $\mathbf{V}$ is an $N \times N$ matrix that is both column and row-orthonormal. The presence of singular values is indicated in matrices $\mathbf{\Sigma}$ and $\mathbf{U}$. A singular value appears as a zero along the diagonal of $\mathbf{\Sigma}$, and has a corresponding zero column in matrix $\mathbf{U}$.

Using SVD, the pseudo-inverse, $\mathbf{A}^+$, of a matrix, $\mathbf{A}$, is given by

$$\mathbf{A}^+ = \mathbf{V}\mathbf{\Sigma}^+\mathbf{U}^T. \tag{2.16}$$

To obtain the pseudo-inverse of $\mathbf{\Sigma}$, each positive diagonal element, σ_j, is replaced by $1/\sigma_j$, or by 0 when $\sigma_j = 0$. [43, 44].

There exist several approaches to obtaining the singular value decomposition of a matrix. These include:

1. a bidiagonalisation (via Householder reductions) and diagonalisation (by QR reductions) method [43],
2. a divide and conquer method [43, 45], and
3. a multiple relatively robust representations (MRRR) method [43, 46, 47].

It is purported that method 2 is faster than method 1, and method 3 is faster than method 2. This speed-up can be attributed to improvements in parallelisation. [43] The implementation used for this problem uses method 1.

Method 1 is also referred to as the Golub-Reinsch method. In [48, 49], Lahabar and Narayanan describe a hybrid CPU/GPU implementation of the Golub-Reinsch method. Their implementation achieves a speed-up of up to 8.2 over an optimised Intel SVD implementation for the CPU.

This work also uses uses a hybrid CPU/GPU Golub-Reinsch implementation of SVD. Specifically, that provided by the Magma package. For this particular decomposition problem, where the matrix to be inverted, $\mathbf{A}$, is square ($M = N$), the Magma SVD routine first performs bidiagonalisation, then diagonalisation [42, 43].

The bidiagonalisation step uses first the Magma routine `magma_*gebrd` and then the CPU LAPACK routine `lapackf77_*orgbr`. The `magma_*gebrd` routine makes extensive use of both the GPU based `magma_*gemv` and the CPU based `blasf77_*gemv`.

On the `magma_*gebrd` routine [42]:

> "`SGEBRD reduces a general real M-by-N matrix A to upper or lower bidiagonal form B by an orthogonal transformation: Q**T * A * P = B.`

> `If m >= n, B is upper bidiagonal; if m < n, B is lower bidiagonal.`"

On the `lapackf77_*orgbr` routine [41]:

> "`SORGBR generates one of the real orthogonal matrices Q or P**T determined by SGEBRD when reducing a real matrix A to bidiagonal form: A = Q * B * P**T. Q and P**T are defined as products of elementary reflectors H(i) or G(i) respectively.`"

The diagonalisation step uses the CPU LAPACK routine `lapackf77_sbdsqr` [41]:

> "`SBDSQR computes the singular values and, optionally, the right and/or left singular vectors from the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B using the implicit zero-shift QR algorithm.`"

Thus, the Magma SVD implementation used here is a hybrid CPU/GPU implementation.

2.4.2.2 Eigenvalue Calculation

The $N \times N$ matrix, $\mathbf{A}$, has an eigenvalue, λ, and a eigenvector, $\mathbf{x}$, where

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

is satisfied. A detailed discussion of the implementation methods for eigenvalue and eigenvector calculations is out of the scope of this work. Press et al. [43], provides a comprehensive introduction to eigenvalue solving algorithms.

2.4.3 Sparse Operations

To calculate the term $\mathbf{W}\mathbf{x}\,(n-1)$ in Equation 2.1, a sparse matrix-matrix multiplication operation was required. This is because the matrix $\mathbf{W}$ is typically sparse (see Section 2.2.2).

To store $\mathbf{W}$, the compressed sparse row (CSR) format was used. This format stores the information about a matrix in three vectors. For an $M \times N$ matrix with NZ non-zero values, these vectors store

1. NZ non-zero values;
2. NZ column indices corresponding to each value; and
3. $M + 1$ values comprising
    a) M pointers indicating the first value value that appears in each row, and
    b) the last value contains the number of non-zeros, NZ. [23]

Previous work has shown that CSR is not the most efficient format in terms of memory coalescing (Section 2.3.1.3), and is best when $NZ > M + 1$. A Hybrid format that combines Ellpack-Itpack with Coordinate format was shown to perform better than CSR, except where the matrix is dense, or near dense, and except where the number of non-zeros per row varies highly. [50]

2.4.4 Bespoke Operations

Three bespoke numerical operations were implemented in this work. These are described in Table 2.3, where a description of the operation is given, along with the equation that requires it.

Table 2.3: The bespoke kernels implemented for this work.

Description	Detail	Required By
Perform a vector sum and hyperbolic tangent.	$\mathbf{x} = tanh\left(\mathbf{v}^1 + \mathbf{v}^2 + \mathbf{v}^3\right)$	Equation 2.1
Generate an $M \times M$, scaled identity matrix.	$\lambda\mathbf{I} = f\left(\lambda, M\right)$	Equation 2.13
Perform a pseudo inverse on a diagonal matrix.	$\mathbf{\Sigma}^{+} = pinv\left(\mathbf{\Sigma}\right)$	Equation 2.16

2.5 Assessing ESN Performance

For this work, the performance of various unique Echo State Network configurations were assessed and compared. The performance measures of interest were execution time and error.

In the case of execution time, *mean* and *standard deviation values* were calculated. These were in-turn used to calculate a *speed-up* measure. In the case of error performance, a *normalised root mean-squared error* was used. *Median*, *upper-*, and *lower-quartile* values were calculated.

2.5.1 Mean & Standard Deviation

Given a set of N measurements $\{t_1, t_2, \ldots, t_N\}$, The mean of these measurements is calculated as

$$\bar{t} = \frac{1}{N}\sum_{i=1}^{N} t_i. \tag{2.17}$$

The standard deviation of the timing measurements are calculated as

$$\sigma_t = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(t_i - \bar{t}\right)^2}. \tag{2.18}$$

2.5.2 Speed-Up

With the timing measurements for both a GPU and a CPU Echo State Network implementation in hand, it was possible to calculate a relative GPU and CPU speed measure. This measure, called *speed-up*, is unitless, and describes a ratio of timing measurements. For the purposes of these experiments, the speed-up was calculated from the perspective of the GPU, i.e. how much faster is the GPU than the CPU when performing a given operation.

Using the mean and standard deviation execution times of a GPU and a CPU ESN configuration, the *mean speed-up* was calculated as

$$\bar{s} = \frac{\bar{t}_{GPU}}{\bar{t}_{CPU}}. \tag{2.19}$$

The *standard deviation speed-up* was calculated as

$$\sigma^s = \sqrt{\left(\frac{\sigma^t_{GPU}}{\bar{t}_{GPU}}\right)^2 + \left(\frac{\sigma^t_{CPU}}{\bar{t}_{CPU}}\right)^2}. \tag{2.20}$$

Here, $\bar{t}_{\text{GPU}}$ and $\bar{t}_{\text{CPU}}$ are the mean GPU and CPU execution times for a given ESN configuration respectively. The σ^t_{GPU} and σ^t_{CPU} values are the standard deviation GPU and CPU execution times. Thus, the GPU can be described as $\bar{s} \pm \sigma^s$ faster than a CPU, where $\bar{s} > 0$. When $\bar{s} < 1$, then the GPU is slower than the CPU.

2.5.3 Normalised Root Mean Square Error

The performance of a trained Echo State Network can be measured using a *normalised root mean squared error* (NRMSE). For this work, the NRMSE was calculated as

$$e(\mathbf{Y}, \mathbf{Y}^{\text{target}}) = \frac{\sum_{i=1}^{N} \left\| \mathbf{y}_i^{\text{target}} - \mathbf{y}_i \right\|^2}{\sum_{i=1}^{N} \left\| \mathbf{y}_i^{\text{target}} - \bar{\mathbf{y}}_i^{\text{target}} \right\|^2}. \tag{2.21}$$

This is the "sum-of-squares" error measurement described by Bishop in [51]. Here, the matrices $\mathbf{Y} = [\ldots, \mathbf{y}_i, \ldots]$ and $\mathbf{Y}^{\text{target}} = \left[\ldots, \mathbf{y}_i^{\text{target}}, \ldots\right]$ have N columns, and represent the actual ESN output, and the target output respectively. The function, $\|\cdot\|$, defines the *2-norm*. This is calculated for each of the N time-steps iterated by the ESN. In [52], the error measurement is said to be "normalised" by the target vector's distance from the mean target, $\bar{\mathbf{y}}_i^{\text{target}}$. As such, the error is measure independent of the range of the target.

Interestingly, Bishop refers to this error a "root mean squared" error, rather than a "normalised root mean squared error". The use of word "root" has been criticised [52], as there is no root given directly in the equation. Although, if one looks at the definition of a 2-norm functioning on an N-size vector, $\mathbf{v}$,

$$\|\mathbf{v}\| = \sqrt{v_1^2 + ... + v_N^2},$$

there is indeed a root present. However, this root is inverted by the enclosing power-of-2. Lukoševičius uses Equation 2.21 enclosed by a square root, and refers to it as a "normalised root mean squared error". This work uses the original definition, given in [51] and Equation 2.21, and refers to it as a "normalised root mean squared error". This acknowledges both the original name used in [51], and the presence of

a normalising denominator as described in [52].

In the special case where the size of the ESN output is 1, then the matrices $\mathbf{Y}$ and $\mathbf{Y}^{\text{target}}$ are $1 \times N$ matrices, $\mathbf{Y} = [\ldots, y_i, \ldots]$ and $\mathbf{Y}^{\text{target}} = \left[\ldots, y_i^{\text{target}}, \ldots\right]$. Thus, the normalised root mean squared error becomes

$$\begin{aligned} e(\mathbf{Y}, \mathbf{Y}^{\text{target}}) &= \frac{\sum_{i=1}^{N} \left\| y_i^{\text{target}} - y_i \right\|^2}{\sum_{i=1}^{N} \left\| y_i^{\text{target}} - \bar{y}_i^{\text{target}} \right\|^2} \\ &= \frac{\sum_{i=1}^{N} \left(y_i^{\text{target}} - y_i \right)^2}{\sum_{i=1}^{N} \left(y_i^{\text{target}} - \bar{y}_i^{\text{target}} \right)^2}. \end{aligned} \tag{2.22}$$

2.5.4 Quartiles

Given a set of N values $\mathcal{E} = \{e_1, e_2, \ldots, e_N\}$, the values are *sorted* in *ascending* order, giving $\mathcal{E}^{\text{sorted}} = \left\{e_1^{\text{sorted}}, e_2^{\text{sorted}}, \ldots, e_i^{\text{sorted}}, \ldots, e_N^{\text{sorted}}\right\}$. The quartiles are the values found at the three positions in $\mathcal{E}^{\text{sorted}}$ that divide $\mathcal{E}^{\text{sorted}}$ into four parts.

The *median* is the central quartile, this is calculated as

$$m = \begin{cases} e_{\lceil N/2 \rceil}^{\text{sorted}} & , \quad N \text{ odd} \\ \frac{1}{2}\left(e_{N/2}^{\text{sorted}} + e_{N/2+1}^{\text{sorted}}\right) & , \quad N \text{ even} \end{cases}.$$

The lower and upper quartiles are "type number 5" quartiles, as described in [53, 54], where the *lower-quartile* is $\hat{Q}_5(0.25)$, and the *upper-quartile* is $\hat{Q}_5(0.75)$.

3 Implementing the Echo State Network

An implementation of the Echo State Network should address three concerns. Firstly, the behaviour of the ESN; secondly, building the ESN; and thirdly training it. The behaviour of the ESN was implemented as described in the original work [3], and as presented in Equations 2.1 and 2.2. To build an ESN, the spectral radius scaling method was implemented. This is described in [13] and summarised in Equation 2.4. The implemented training method is based on Tikhonov regularisation as described in [15, 2] and Equation 2.13.

The resulting implementation extends upon previous work performed in MATLAB [22]. In this work, we described a MATLAB and AccelerEyes Jacket implementation of the ESN on a low-end graphics card. Given the results, we recommended a lower level language implementation. To this end, two shared C^{++} libraries `libesnmath.so` and `libgpuesn.so` were constructed. The remainder of this section will discuss the design of these libraries and the tools used to build them. First the responsibilities and dependencies of the libraries are defined, this is followed by a basic example of using the libraries. The details of the libraries then follow, including the numerical operations required by each library, and the co-ordination of these operations.

3.1 High Level Design and Deployment

The implementation comprises two libraries, each of which interfaces with libraries distributed alongside the Nvidia drivers, and the Magma libraries. One library, `libgpuesn.so`, is compiled using `g++` only; the other, `libesnmath.so`, is compiled using both `nvcc` and `g++`. The relationship between these libraries is described in Figure 3.1.

3.1.1 The `libesnmath.so` Library

The `libesnmath.so` library is a shared library that contains several bespoke Cuda kernels, and interfaces with the Nvidia Cuda and Magma libraries. These kernels are used alongside standard BLAS and LAPACK operations to implement the Echo State Network, and the Tikhonov regularisation algorithms. Descriptions of the three kernels are given in Table 2.3.

The `libesnmath.so` library is compiled using Nvidia's `nvcc-4.2` and `g++-4.7.2`. At compilation time, a flag specifying compute capability is passed to `nvcc` (see Section 2.3.2.1). This flag, "`-gencode arch=compute_20,code=compute_20`", ensures that device-non-specific Parallel Thread Execution (PTX) code is generated at compilation time. In this case, the generated PTX code requires a device of compute capability 2.0 or higher. Upon first execution of this PTX code on a given machine, it is compiled "just in time", producing binaries specific to that machine. A compute capability of 2.0 is specified, as the library is designed to use parallel kernel execution. This feature is available only on some devices of compute capability of 2.0 or higher (see Section 2.3.1.2).

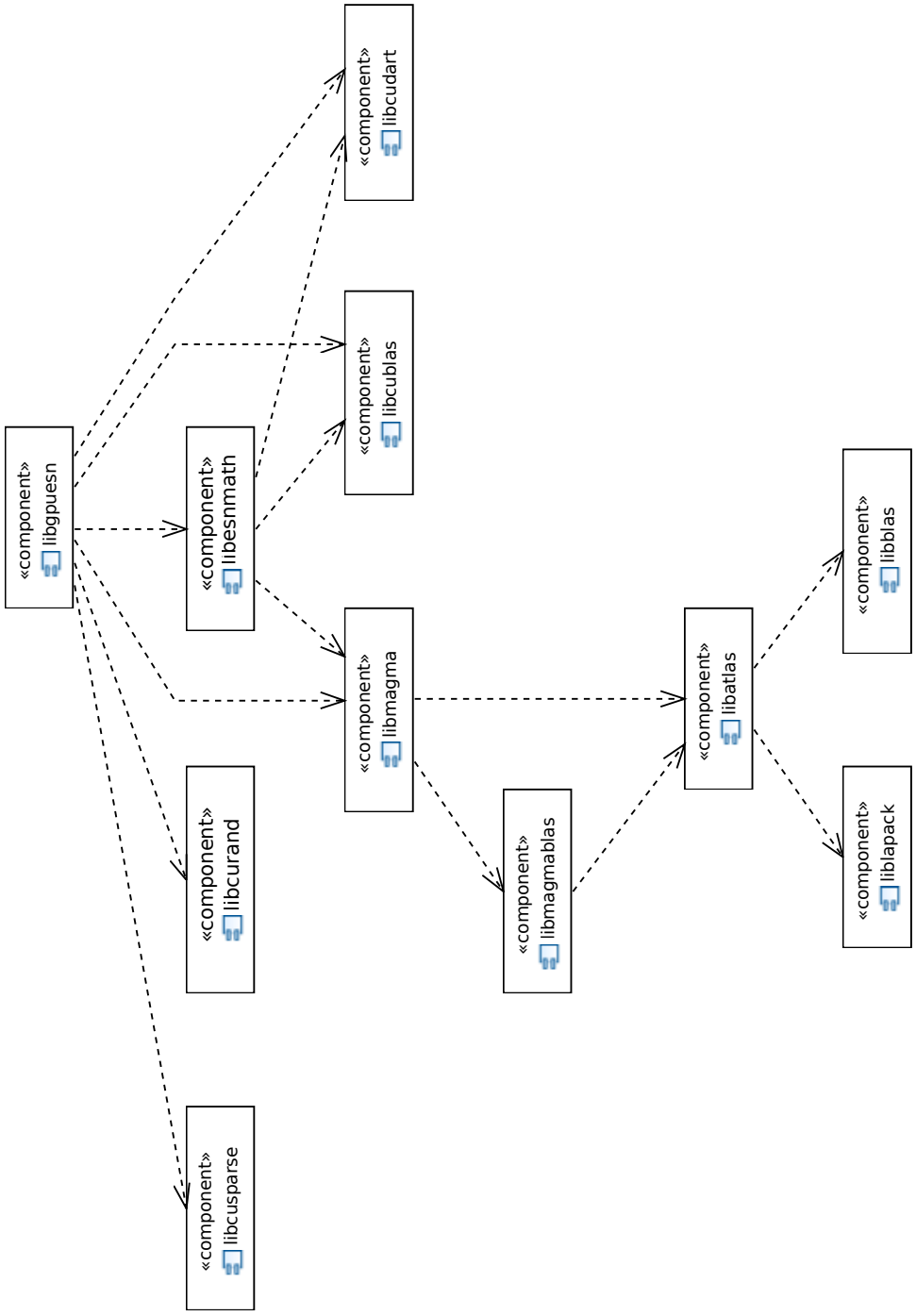

Figure 3.1: A UML component diagram that describes the Echo State Network components, and their relationship to the Nvidia, Magma, and reference BLAS and LAPACK libraries.

3.1.2 The `libgpuesn.so` Library

The `libgpuesn.so` library is a shared library that contains the Echo State Network and Tikhonov regularisation implementations. It makes use of the aforementioned `libesnmath.so`, Magma, and Nvidia's linear algebra libraries. This library was compiled from pure C^{++} code (i.e. C^{++} code without Nvidia language extensions) using the latest version of `g++` on the development computers (currently `g++-4.7.2`). This library interfaces with the aforementioned `libesnmath.so`, and Nvidia's `libcudart.so`, `libcusparse.so`, `libcurand.so`, Magma, and `libcublas.so` libraries.

This library implements the Echo State Network and Tikhonov regularisation algorithms. To access these, users can create C^{++} objects from classes

- `EsnBuilder<T>`,
- `EchoStateNetworkCublas<T>`, and
- `EsnTrainerCublas<T>`.

Where `T` is the template-type `double` or `float`.

3.2 Using the Libraries

Listings 3.1 – 3.4 demonstrate interfacing with `libgpuesn.so` to build and train and Echo State Network using C^{++}. Listing 3.1 describes the inclusions and namespaces required to build and train an ESN. Listing 3.2 describes the configuration variables required to build an ESN. Listing 3.3 lists the training data required to train an ESN. Finally, Listing 3.4 gives the code required to declare, build and train and ESN.

Listing 3.1 Building and training an Echo State Network – the required inclusions.

```
#include <EsnBuilder.h>
#include <EchoStateNetworkCublas.h>
#include <EsnTrainerCublas.h>
//...
using namespace Esn;
//...
```

Listing 3.2 Building and training an Echo State Network – the required configuration variables.

```
// Initialise Curand, Cusparse, and Cublas handles
curandGenerator_t curandGenerator;
cusparseHandle_t cusparseHandle;
cublasHandle_t cublasHandle;
//...
// Initialise desired ESN properties
// - the dimensions of the ESN
uint inputSize, outputSize, reservoirSize;
// - the spectral radius, rho(W), of the reservoir
T spectralRadius;
// - the proportion of non-zero values in the reservoir
//   [0,1]
float connectivity;
// - whether the ESN has output feedback
bool hasOutputFeedback;
//...
```

Listing 3.3 Building and training an Echo State Network – the variables required to train the ESN.

```
// Prepare training data
// - the input training data
//   this must be in COLUMN-major format
T *trainDataIn;
uint trainDataInCount
// - the target training data
//   this must be in COLUMN-major format
T *trainDataOut;
uint trainDataOutCount;
// - the Tikhonov regularisation factor
T *regFact
// - the proportion of timesteps to discard before
//   performing Tikhonov regularisation [0,1].
double discardProportion;
//...
```

Listing 3.4 Building and training an Echo State Network.

```
// Build and train an ESN
EsnBuiler<T> *builder = new EchoStateNetwork<T>(
   curandGenerator,
   cusparseHandle,
   cublasHandle);
EchoStateNetwork<T> *esn = builder->GenerateRandomEchoStateNetwork(
   inputSize,
   outputSize,
   reservoirSize,
   spectralRadius,
   connectivity,
   hasOutputFeedback);
EsnTrainer<T> *trainer = new EsnTraininerCublas<T>(
   cublasHandle,
   cusparseHandle);
trainer->Train(
   esn,
   trainDataIn, trainDataInCount,
   trainDataOut, trainDataOutCount,
   regFact,
   discardProportion);
```

3.3 Numerical Operations

This section contains a summary of the numerical operations used to implement the Echo State Network, and training via Tikhonov regularisation. More specifically, the operations required to build, run, and train and ESN. For each algorithm, the mathematical expression is reproduced, alongside a description, and a table of operations required. The operations used are provided by either an Nvidia library, the Magma library, or the bespoke kernels.

3.3.1 ESN Building

The implementation uses the reservoir scaling method described in Section 2.2.3 and Equation 2.4,

$$\mathbf{W} = \frac{\rho \mathbf{W}_{\text{rand}}}{max\left(|\lambda\left(\mathbf{W}_{\text{rand}}\right)|\right)}.$$

This describes scaling a random matrix, $\mathbf{W}_{\text{rand}}$, to obtain a matrix with a desired spectral radius, ρ. To implement this, LAPACK and BLAS operations, and a pseudo-random number generator were used. These are listed in Table 3.1.

3.3.1.1 Generating $\mathbf{W}_{\text{rand}}$

The matrix $\mathbf{W}_{\text{rand}}$ is generated using a uniform pseudo-random number generator. The generator is capable of drawing values from the uniform distribution over the range $(0, 1]$. The resulting values are scaled to achieve a range of $(-1, 1]$. The number of non-zeros (NNZ) generated depends on the size of the reservoir (N),

and the density (σ, where $0 < \sigma \leq 1$), of the matrix:

$$NNZ = \begin{cases} \lceil \sigma N \rceil & , \quad \sigma N < 1 \\ round\,(\sigma N) & , \quad \sigma N \geq 1 \end{cases} .$$

3.3.1.2 Scaling $\mathbf{W}_{\text{rand}}$

The implementation of Equation 2.4 first calculates the equation's denominator. Given that $\mathbf{W}_{\text{rand}}$ is random and sparse, there is a chance that the denominator will be zero. In this case, the matrix is discarded and a new random matrix generated. The denominator is calculated using a single eigenvector calculation and a maximum magnitude operation. Once a non-zero denominator is found, $\mathbf{W}_{\text{rand}}$ is scaled using a vector scaling operation. A summary of the operations used, and their providing libraries, is given in Table 3.1.

Table 3.1: The operations required for reservoir scaling operation. This assumes that the denominator in Equation 2.4 is non-zero on the first attempt.

Operation	Count	Implementation	Library
Generate uniform random numbers	1	curandGenerateUniform(Double)	Curand
Determine eigenvector	1	geev	Magma
Determine maximum magnitude	1	amax	CuBLAS
Scale vector	1	scal	CuBLAS

3.3.2 ESN Execution

The behaviour of the Echo State Network can be summarised in Equations 2.1 and 2.2. These are reproduced here for convenience. Equation 2.1,

$$\mathbf{x}(n) = f\left(\mathbf{W}^{\text{in}}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1) + \mathbf{W}^{\text{ofb}}\mathbf{y}(n-1)\right),$$

describes the calculation of the reservoir state at time n. Equation 2.2,

$$\mathbf{y}(n) = f_{\text{out}}\left(\mathbf{W}^{\text{out}}\begin{bmatrix}\mathbf{u}(n)\\ \mathbf{x}(n)\end{bmatrix}\right),$$

describes the calculation of the ESN output at time n. It is important to note that the second term in Equation 2.1 involves the sparse matrix $\mathbf{W}$. The matrices in the first and third terms may also be sparse, but typically, they are dense.

Available to the Nvidia Cuda GPU programmer are level 1, 2, and 3 BLAS routines, Sparse routines, trigonometric operations, memory management operations, and an ability to create bespoke kernels. All four tools were used in the implementation of these equations.

3.3.2.1 State Calculation

The input, $\mathbf{u}$, in Equation 2.1 requires a copy from host to GPU device memory. Once this is on the GPU memory, all other operations are performed there.

Equation 2.1 has two dense matrix-vector multiplications, one sparse matrix-vector multiplication. These are performed in parallel, into three temporary vectors. The remaining sum and hyperbolic tangent operations are performed in a single bespoke kernel. Where each element of the output vector is calculated in its own thread. A summary of the operations used, and their providing libraries, is given in Table 3.2.

Table 3.2: The operations required for ESN reservoir state calculation.

Operation	Count	Implementation	Library
Host to GPU memory	1	cudaMemcpy	Cuda
Dense matrix-vector multiplication	2	gemv	CuBLAS
Sparse matrix-vector multiplication	1	spmv	Cusparse
Vector sum and element-wise *tanh*	1	bespoke kernel	–

3.3.2.2 Output Calculation

The input, $\mathbf{u}$, and reservoir state, $\mathbf{x}$, vectors are stacked in a temporary vector using two parallel GPU-side memory copy operations. Following this, a dense matrix-vector multiplication is performed on the output weight matrix, $\mathbf{W}^{\text{out}}$, and the temporary vector. In this implementation $f_{\text{out}}(\mathbf{v}) = \mathbf{v}$, thus no further processing is required (i.e. $f_{\text{out}}(\cdot)$ is an identity function). A summary of the operations used, and their providing libraries, is given in Table 3.3.

Table 3.3: The operations required for ESN output calculation.

Operation	Count	Implementation	Library
GPU-side memory copy	2	cudaMemcpy	Cuda
Dense matrix-vector multiplication	1	gemv	CuBLAS

3.3.3 ESN Training

This implementation of ESN training uses the offline Tikhonov regularisation method. This is described in Equation 2.13,

$$\mathbf{W}_{\text{out}} = \mathbf{Y}_{\text{target}}\mathbf{X}^T\left(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I}\right)^{-1}.$$

Note that this equation includes the inversion of the square matrix, $\left(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I}\right)$. To do this, this implementation uses singular value decomposition (SVD). A brief description of SVD is given, then the details of the implementation follow.

3.3.3.1 Numerical Operations

To implement Equation 2.13, first the inversion term, $\left(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I}\right)$, is calculated. This requires generating a scaled identity matrix, a matrix-matrix multiplication, and vector addition. The inversion is performed using the singular value decomposition method described in Section 2.4.2.1. The inversion thus requires an SVD operation, a diagonal-matrix pseudo-inversion operation, and two matrix-matrix operations. The resulting inverse is then used in two final matrix-matrix multiplication operations to obtain $\mathbf{W}^{\text{out}}$. Table 3.4 summarises these operations, and the libraries that were used.

Table 3.4: The operations required for the Tikhonov regularisation operation.

Operation	Count	Implementation	Library
Generate scaled identity matrix	1	bespoke kernel	–
Vector addition	1	axpy	CuBLAS
Singular value decomposition	1	gesvd	Magma
Diagonal matrix pseudo-inverse	1	bespoke kernel	–
Matrix-matrix multiplication	5	gemm	CuBLAS

3.4 Program Design

To create a GPU Echo State Network, the operations described in Section 3.3 were coordinated to take advantage of the concurrent execution model of the GPU. This section details the techniques used.

3.4.1 ESN Memory Management

Efforts were made to reduce the time spent loading and fetching memory during Echo State Network execution. To do this, at Echo State Network execution time, weights are transferred to and held on the GPU device memory for the duration of the ESN's existence. In addition to this, memory transfers at ESN runtime are performed via page-locked memory, ESN inputs via write-combined page-locked memory, and outputs via default page-locked memory.

3.4.2 ESN Execution

A double buffering approach was taken to the Echo State Network implementation. At ESN execution time, while one ESN output is calculated, a simultaneous GPU device memory load and fetch are taking place. More specifically, the outputs from the last ESN calculation are fetched from device memory, and the inputs for the next ESN calculation are loaded. These parallel calculate, load, and fetch operations are facilitated by the use of Cuda streams (see Section 2.3.1.2) and double buffers (see Section 2.3.1.4).

3.4.2.1 Input/Output Double Buffers

A pair of double buffers, or *ping-pong* buffers are used for Echo State Network input and output. These reside in GPU device memory. For example, while the ESN works on the *ping* input buffer to produce the outputs written to the *ping* output buffer, the *pong* input buffer is loaded ready for the next execution, and data from the last *pong* output buffer is fetched from GPU device memory. This detailed in Section 3.4.2.4.

3.4.2.2 Cuda Streams

The pipelining effect is achieved using Cuda streams created during the construction of the ESN. Modern Nvidia Cuda GPU architectures are able to run these streams in parallel. Several streams are created and used during the lifetime of an ESN. These are streams for

1. loading inputs (one each for ping and pong executions);

2. fetching outputs (one each for ping and pong executions);

3. calculating the first matrix-vector operation, $\mathbf{W}^{\text{in}}\mathbf{u}(n)$, the vector-sum and hyperbolic-tangent operation $tanh(\mathbf{W}^{\text{in}}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1) + \mathbf{W}^{\text{ofb}}\mathbf{y}(n-1))$, and the output matrix-vector operation, $\mathbf{W}^{\text{out}}[\mathbf{u}(n)|\mathbf{x}(n)]$;

4. calculating the second matrix-vector operation, $\mathbf{W}\mathbf{x}(n-1)$; and

5. if using output feed-back, calculating the third matrix-vector operation, $\mathbf{W}^{\text{ofb}}\mathbf{y}(n-1)$.

Here, 1 and 2 can be thought of as *memory transfer streams*; 3, 4, and 5 as *kernel execution streams*. For this implementation, best performance is achieved with a Cuda device that can execute memory transfers and kernel executions in parallel streams, and can also execute different kernels in parallel streams. Sections 3.4.2.3 and 3.4.2.4 describe this use of streams in more detail: Section 3.4.2.3 details stream usage when executing a single time-step of the Echo State Network, this is shown without the input and output memory transfers. Section 3.4.2.4 describes how memory transfers, both input and output, and Echo State Network execution are performed in parallel.

3.4.2.3 An ESN Time Step

In Figure 3.2, the behaviour of the `EchoStateNetworkCublas<T>` class's `DoEsnCalculationStep(...)` function is shown.[1] Here, four Cuda streams and a single host thread are illustrated. The host thread, labelled *libgpuesn*, controls the calculation. The streams labelled *calculation 1*, *calculation 2*, and *calculation 3* perform linear algebra operations, and the stream labelled *memory set* performs a GPU device memory transfer. The *libgpuesn* thread begins with asynchronous calls to initiate

- the "First matrix multiplication", $\mathbf{W}^{\text{in}}\mathbf{u}(n)$, on the *calculation 1* stream;
- the "Second matrix multiplication", $\mathbf{W}\mathbf{x}(n-1)$, on the *calculation 2* stream;
- and the "Third matrix multiplication", $\mathbf{W}^{\text{ofb}}\mathbf{y}(n-1)$, on the *calculation 3* stream.

The *libgpuesn* thread then waits for these multiplications to complete concurrently. Once completed, the *libgpuesn* thread initiates a synchronous call to "Sum and tanh", $tanh(\mathbf{W}^{\text{in}}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1) + \mathbf{W}^{\text{ofb}}\mathbf{y}(n-1))$, on the *calculation 1* stream; and following this, a synchronous call to the "Output matrix multiplication",

$$\mathbf{W}^{\text{out}} \begin{bmatrix} \mathbf{u}(n) \\ \mathbf{x}(n) \end{bmatrix}.$$

Finally, the *libgpuesn* thread initiates a GPU device memory copy to store the output vector for use in the next time step ("Store output for next execution"). The `DoEsnCalculationStep(...)` function exits before the memory copy is complete, this allows the caller to perform other activities in parallel with the copy. For the caller to synchronise with this memory copy, `DoEsnCalculationStep(...)` returns an

[1] The execution times are not given to scale, rather they illustrate an approximate timing of the interactions between the host thread, and the GPU device streams.

`ITask*`, the caller can then use the `ITask::WaitOnComplete()` function to synchronise.

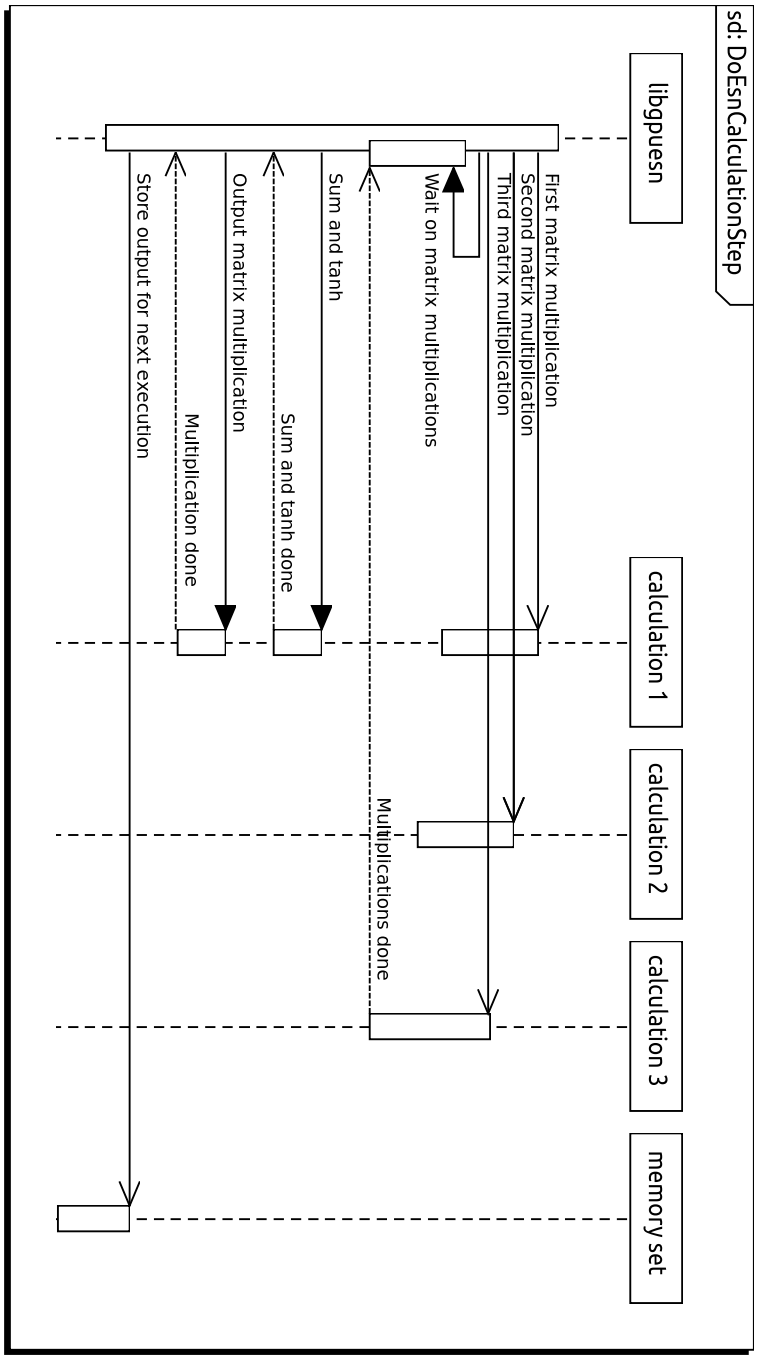

Figure 3.2: A UML sequence diagram describing the calculation of a single Echo State Network step (see Footnote 1 on the preceding page).

3.4.2.4 An ESN Time Step with Double Buffered Memory Transfers

Figure 3.3 describes the double buffering of memory transfers and Echo State Network calculations during ESN execution (see Footnote 1 on page 56). Here, only a snapshot of a *ping* execution is shown while *pong* buffers are prepared for the next execution. Four Cuda streams and a single host thread are illustrated. The host thread, labelled *libgpuesn*, controls the memory transfers and calculation. The streams labelled *set input pong*, *fetch output pong*, *set input ping*, and *memory set* perform memory transfers.

This snapshot begins when *libgpuesn* is waiting for the previous pong execution to complete on the *memory set* stream ("Wait for pong store last output to complete"). This is the final task performed by the call to `DoEsnCalculationStep(...)` (as described in Section 3.4.2.3) initiated in the previous pong step. Once this has completed, *libgpuesn* can then asynchronously

1. fetch the output generated in the previous pong step ("Fetch pong output") on the *fetch output pong* stream, and

2. prepare the pong input buffer for the next pong execution ("Set pong input") on the *set input pong* stream.

After initiating these memory transfers, *libgpuesn* then

1. ensures that its current ping input is ready ("Wait for ping input to load" on the *set input ping* stream); and then

2. begins the ESN time step calculation, calling `DoEsnCalculationStep(...)` as described in Section 3.4.2.3 ("Ping DoEsnCalculationStep").

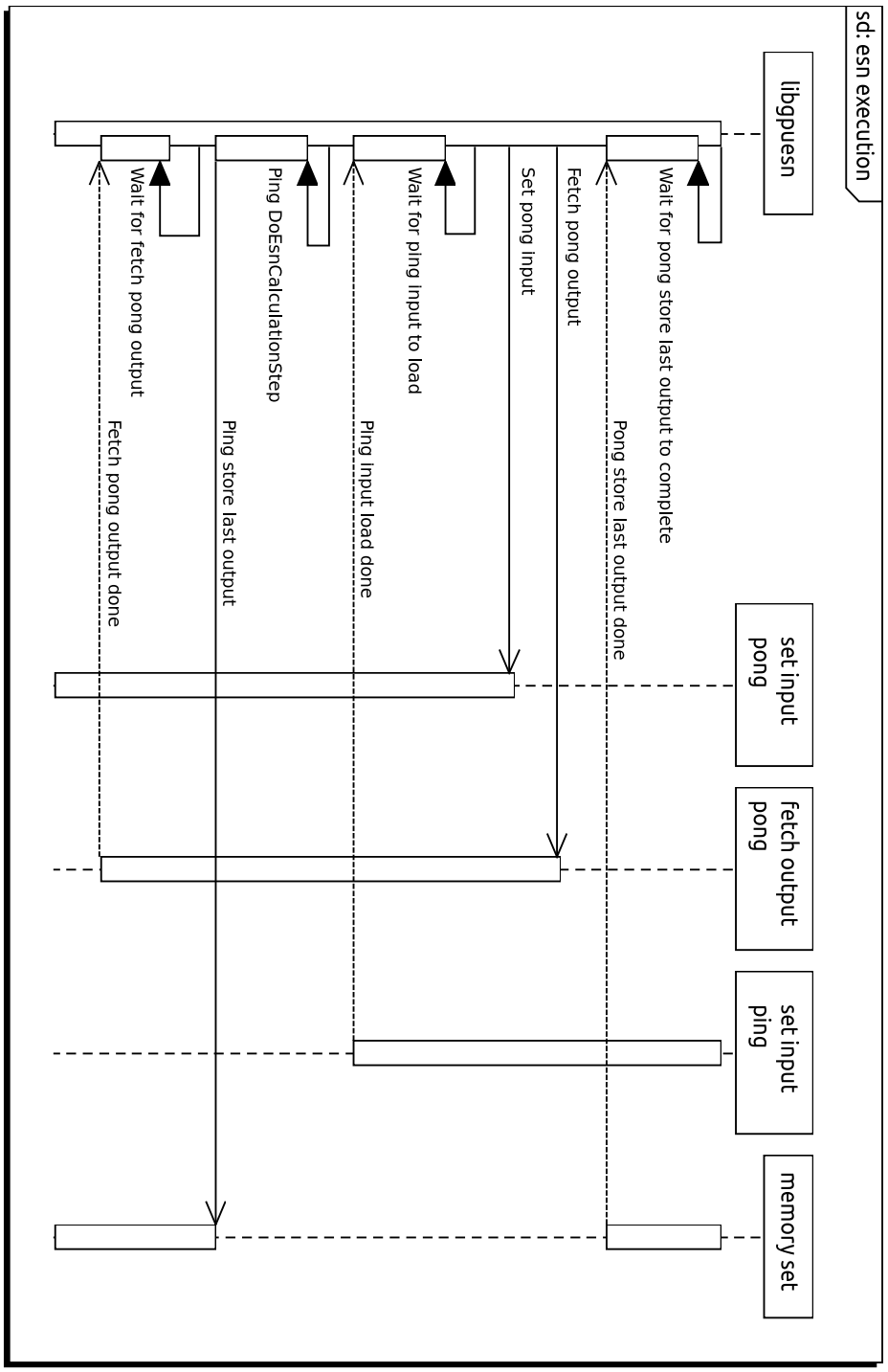

Figure 3.3: A UML sequence diagram describing parallel load, fetch, and execute operations during Echo State Network execution (see Footnote 1 on page 56).

The last task performed during the execution of `DoEsnCalculationStep(...)` is to asynchronously store the last ESN output for use in the next ESN calculation step ("Ping store last output" on the *memory set* stream). In parallel with this, *libgpuesn* waits for the previously initiated pong output fetch to complete ("Wait for fetch pong output" on the *fetch output pong* stream), then the sequence begins again. Of course, in the next sequence, pong buffers and streams are replaced by ping buffers and streams, and *vice versa*.

3.4.3 Training the ESN via Tikhonov Regularisation

In Figure 3.4, the behaviour of the `EsnTrainerCublas<T>` class's `DoLinearRegression(...)` function is shown (see Footnote 1 on page 56). Here, three Cuda streams and a single host thread are illustrated. The host thread, labelled *libgpuesn*, controls the calculation. The streams labelled *calculation 1*, *calculation 2*, and *calculation 3* perform linear algebra operations. The *libgpuesn* thread begins with asynchronous calls to initiate

- the $\mathbf{X}\mathbf{X}^T$ matrix-matrix multiplication ("First gemm"), on the *calculation 1* stream;
- the $\lambda\mathbf{I}$ identity generation operation ("Generate ident.") on the *calculation 2* stream; and
- the $\mathbf{Y}_{\text{target}}\mathbf{X}$ matrix-matrix operation ("Second gemm") on the *calculation 3* stream.

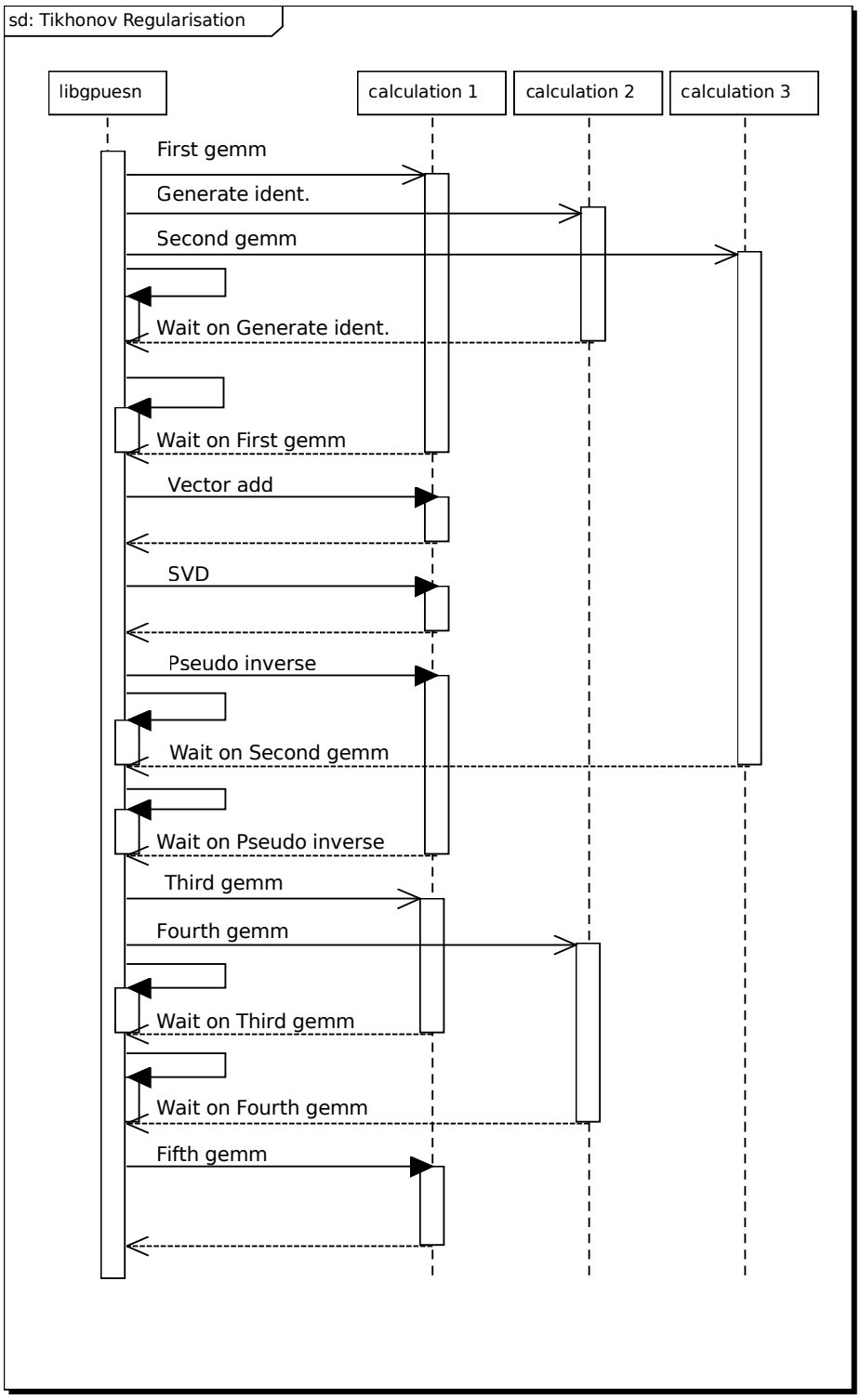

Figure 3.4: A UML sequence diagram describing parallel numerical operations during Tikhonov regularisation (see Footnote 1 on page 56).

The *libgpuesn* thread then waits for streams *calculation 1* and *calculation 2* to finish. Once these streams are idle, the *libgpuesn* thread initiates the $\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I}$ vector-addition ("Vector add"). Upon completion of this addition, the inversion process begins. This process uses singular value decomposition to perform the inversion, as described in Section 2.4.2.1. The *libgpuesn* thread initiates the $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = svd\left(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I}\right)$ operation ("SVD") on the *calculation 1* stream. Once this is complete, the *libgpuesn* thread then executes the pseudo-inverse operation, $\mathbf{\Sigma}^+ = pinv\left(\mathbf{\Sigma}\right)$ ("Pseudo inverse"), on the *calculation 1* stream. The *libgpuesn* thread then waits on the completion of the $\mathbf{Y}_{\text{target}}\mathbf{X}$ matrix-matrix operation on the *calculation 3* stream. When this is complete, the *libgpuesn* thread then executes the first of the final matrix-matrix operations, $\left(\mathbf{Y}_{\text{target}}\mathbf{X}\right)\mathbf{V}$ ("Third gemm") on the *calculation 2* stream. The *libgpuesn* thread then waits on the completion of the pseudo-inversion operation on the *calculation 1* stream. Once complete, the matrix-matrix multiplication operation, $\mathbf{\Sigma}^+\mathbf{U}^T$ ("Fourth gemm") is initiated on the *calculation 1* stream, then the *libgpuesn* waits on both the *calculation 1* and *calculation 2* streams. Once these operations are completed, the libgpuesn thread can then initiate the final matrix-matrix multiplication, $\mathbf{W}^{\text{out}} = \left(\mathbf{Y}_{\text{target}}\mathbf{X}\mathbf{V}\right)\left(\mathbf{\Sigma}^+\mathbf{U}^T\right)$ ("Fifth gemm"). Where the result is written directly to the portion of GPU-device memory that stores the ESNs output weights.

4 Performance and Behaviour of the GPU ESN

The size of an Echo State Network's reservoir, and the amount of data to use to train it are important considerations when designing an ESN. The size of the reservoir determines the complexity of the ESN, and the complexity of the problems it can learn [3, 13]. The amount of training data required is dependent on the complexity of the problem (see Section 2.2.3). As such, the experiments presented in this chapter were devised to examine the behaviour of the GPU Echo State Network, when the ESN's reservoir size and the amount of training data is varied. The first set of experiments looked at GPU execution time relative to CPU execution time. The second set of experiments tested the GPU ESN on several prediction problems, and used time-series previously examined in the ESN literature.

The first set of experiments aimed to explore the relationship between an Echo State Network's execution time and its reservoir size. These experiments extended upon previous work [22]. In this earlier work, we described a MATLAB and AccelerEyes Jacket implementation of the ESN (Equations 2.1 and 2.2) on a low-end graphics card. This MATLAB implementation was run on both a CPU (2.4 GHz Core 2 Duo) and a GPU (the NVIDIA GeForce 9400M). The GPU gave a speed-up of 2 for a reservoir size of 1800. Given these results, there was evidence that larger speed-ups

could be achieved with a lower level language (e.g. C/C^{++}) implementation on a higher-end graphics card.

The experiments presented here performed a similar comparison. Here, the GPU ESN implementation described in Chapter 3 was compared with a CPU implementation. The operations used when executing an ESN (Equations 2.1 and 2.2) were examined. The operations required to execute the Tikhonov regularisation (TR) algorithm used in offline training (Equation 2.13) were also examined. The experiments investigated the relationship between execution time and reservoir size. In the TR case, the relationship between execution time and the amount of training samples was also examined.

The second set of experiments aimed to show that the GPU implementation can behave as a multi-time-step predictor. Here, the Echo State Network was trained to accept a time-series, and to predict the value of this time-series multiple samples into the future. The ESN's behaviour was tested against three time-series. The first, a simple sinusoid [3, 13]; the second, the well known Mackey-Glass problem [3, 13, 5]; and the third, the multiple superimposed oscillator (MSO) problem [55].

The following three sections describe these two experiments, and begin with a description of the hardware used in both cases.

4.1 Test Machine Parameters

The test platform chosen comprised an Intel i7-980 CPU and an Nvidia GTX480. Both representative of high-end commodity processors of their class. The implementations used *all 4 cores* of the CPU, and *all 480 cores* of the GPU. A single core CPU implementation was considered, however, this was seen as an unfair comparison. Multi-core CPUs are the norm today, and it is rare to find a scientific

computing user that uses a single-core CPU. A summary of the relevant test platform specifications are given in Table 4.1.[1,2]

Table 4.1: Selected CPU and GPU parameters.

	Intel Core i7-920	Nvidia GTX480
Core count.	4	480
Thread count.	8	23,040
Core clock speed.	2.67 GHz	1.401 GHz
Warp size.	–	32
Concurrent kernels.	–	true
Memory.	6 GiB	1.5 GiB
Memory clock speed.	1.066 GHz	1.848 GHz
Shared memory per block.	–	48 KiB
PCI bus speed.	–	2.5 GiT/s

4.2 Echo State Network Speed Comparison

The speed performance of the GPU Echo State Network implementation was compared with a CPU implementation. Two components of the Echo State network execution were examined. The first, the execution of the Echo State Network as described in Equations 2.1 and 2.2. The second, training the network using Tikhonov regularisation, as described in Equation 2.13. This section begins by defining the two problems that were examined, the experimental method follows, and finally the results are presented. A conference article based on these results has been since accepted into the 12$^{\text{th}}$ International Conference on Artificial Intelligence and Soft-Computing, and will be published by Springer in their Lecture Notes in Artificial Intelligence series [56].

[1] GiT/s (gibitransfers per second) is equivalent to gibibytes per second and includes PCI protocol overheads.

[2] Host-side random access memory compared with GPU-side global memory.

4.2.1 ESN Execution

Echo State Network execution refers to the operations required to calculate the state of an ESN's reservoir (Equation 2.1), and the ESN's outputs (Equation 2.2). These operations are used when executing a trained ESN over data. In the case of offline training, Equation 2.1 is used to capture a reservoir state history (see Section 2.2.5.1).

For this experiment, the problem can be stated as follows: Given a GPU and a CPU implementation of Equation 2.1,

$$\mathbf{x}(n) = f\left(\mathbf{W}^{\text{in}}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1) + \mathbf{W}^{\text{ofb}}\mathbf{y}(n-1)\right),$$

and Equation 2.2,

$$\mathbf{y}(n) = f_{\text{out}}\left(\mathbf{W}^{\text{out}}\begin{bmatrix}\mathbf{u}(n)\\ \mathbf{x}(n)\end{bmatrix}\right),$$

execute these equations over E time-steps. When performing this over a range of reservoir sizes, N, at which points is the GPU faster than the CPU?

4.2.2 Tikhonov Regularisation

Tikhonov regularisation (Equation 2.13) is an operation that can be used when performing offline training. It is used in the last step of training, after a history of reservoir states (Equation 2.10) has been captured using Equation 2.1. The operation uses the captured state history and training data to calculated the output weights, $\mathbf{W}_{\text{out}}$, of the Echo State Network (see Section 2.2.5.1).

For this experiment, the problem can be stated as follows: Given a GPU and a CPU

implementation of Equation 2.13,

$$\mathbf{W}_{\text{out}} = \mathbf{Y}_{\text{target}}\mathbf{X}^T \left(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I}\right)^{-1},$$

execute this equation over a range of state history matrix sizes, Equation 2.10,

$$\mathbf{X} = \begin{bmatrix} \mathbf{u}_{\text{t}}\left(T_0\right) & \dots & \mathbf{u}_{\text{t}}\left(T\right) \\ \mathbf{x}_{\text{t}}\left(T_0\right) & \dots & \mathbf{x}_{\text{t}}\left(T\right) \end{bmatrix}.$$

Here, $\mathbf{X}$ is an $(N + K) \times (T - T_0)$ matrix, where N is the reservoir size, K is the input size, and $T - T_0$ is the number of samples used in training. When training Echo State Networks over a range of reservoir sizes, and for differing training data-set sizes, at which points is the GPU faster than the CPU?

4.2.3 Experimental Configuration

In the Echo State Network execution case, the experiment was configured to examine the relationship between execution time and reservoir size. In the Tikhonov regularisation (TR) problem, the experiment was configured to examine the relationship between the size of the state-history matrix, $\mathbf{X}$ (Equation 2.10), and execution time. To facilitate a GPU/CPU comparison, the ESN and TR algorithms were implemented for the multi-core CPU. The experiment was configured to reduce the impacts of just-in-time compiler optimisations, and unexpected load-imbalances on the test machine.

4.2.3.1 Multiple Timing Measurements

Multiple timing measurements were used for each ESN configuration. This was done to reduce the impact of sudden and unexpected test-computer loads on a

measurement. These loads can occur due to processing tasks performed by the operating system, and are out of the control of the test computer user.

4.2.3.2 Warm-up Time

To reduce the impact of *just-in-time*-compiled components (see Section 2.3.2.1) and other run-time optimisations on the timing measurements, a "warm-up" period was used. This warm-up was done for each Echo State Network configuration, before the timing measurements began. During this warm-up period, the operations under observation were performed and timed, and the timings were discarded.

4.2.3.3 CPU Implementation

To examine the problems presented in Section 4.2, required a CPU implementation of the ESN and Tikhonov training equations. This implementation was performed using GNU-Octave [54], a high-level interpreted linear algebra language. This interfaced with the Atlas library [57], also known as the automatically tuned linear algebra software library. This in-turn interfaced with the reference Fortran77 BLAS [40] and LAPACK [41] linear algebra routines.

One could argue that using an interpreted language for the CPU implementation gives an unfair speed advantage to the compiled GPU implementation. This should be taken into consideration when viewing the results that follow. A CPU implementation using a compiled or just-in-time compiled language is considered for future work (see Section 5.1.6).

4.2.3.4 Numerical Input & Operations

For this experiment, speed was the major consideration. As such, pseudo-random data was used as inputs during these experiments. The input values for the CPU and GPU implementations were generated using different pseudo-random number generators. Therefore, different inputs are used in both cases.

Similarly, the Echo State Networks were generated pseudo-randomly, using different pseudo-random number generators in the CPU and GPU cases. Thus, the numerical operations performed in each case will not be identical, which could perhaps hinder the comparability of the two cases. This could be addressed in future work (see Section 5.1.8).

4.2.3.5 Experimental Parameters

The variables of interest in these problems are execution time, floating-point precision, reservoir size, and training sample count. To ensure comparability across all measurements taken, the remaining parameters were fixed to arbitrary values. A summary of the values used is given in Table 4.2.

It should be noted that the value assigned to both the ESN input size, K, and the ESN output size, L, is 16. One could argue that configuring the ESN to accept and output vectors, rather than a scalar, favours the GPU, because the GPU is optimised for vector and matrix operations. As such, this should be taken into consideration when viewing the results. An experiment that also considers ESN input and output sizes is considered for future work (see Section 5.1.7).

Table 4.2: Echo State Network and Tikhonov regularisation speed tests. The experimental parameters.

Variable	ESN Values	TR Values
Hardware.	{Intel i7-980, Nvidia GTX-480}	
Calculation precision.	{double, single}	
Number of warm-ups per ESN/TR configuration.	20	
Number of timing measurements per ESN/TR configuration.	20	
ESN reservoir size, N (size of $\mathbf{x}$, Eq. 2.1).	$\{2^4, 2^5, \ldots, 2^{11}\}$	
ESN execution time-steps, E.	$\{2^4, 2^5, \ldots, 2^{16}\}$	
ESN input size, K.	2^4	
ESN output size, L.	2^4	
Training samples (number of columns, $T-T_0$, in $\mathbf{X}$. See Eq. 2.10)	–	$\{2^4, 2^5, \ldots, 2^{16}\}$
ESN output feedback (presence of $\mathbf{W}^{\text{ofb}}$ term in Eq. 2.1).	present	–
ESN reservoir connectivity (proportion of non-zero values in $\mathbf{W}$, Eq. 2.1).	0.1	–
ESN reservoir spectral radius, $\rho(\mathbf{W})$ (Eq. 2.3).	0.9	–
Tikhonov regularisation factor, λ (Eq. 2.13).	–	0.1

4.2.4 ESN Execution Speed

The results of the Echo State Network execution timing measurements are given in Figure 4.1 and Table 4.3. Figure 4.1 plots the raw timing measurements taken, Table 4.3 gives the mean and standard deviation GPU speed-up as defined in Section 2.5.2[3].

[3]When viewing the results, note the comments on CPU implementation (see Section 4.2.3.3), and the size of ESN inputs and outputs (see Section 4.2.3.5).

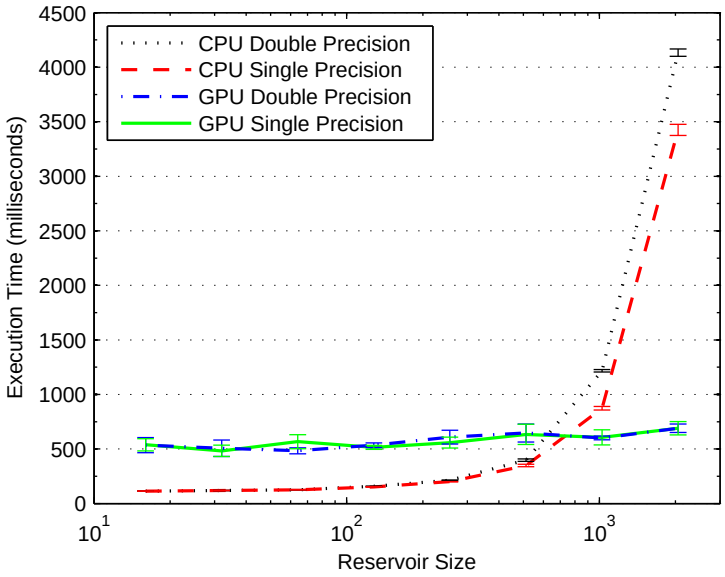

Figure 4.1: The Echo State Network, mean (Equation 2.17) and standard deviation (Equation 2.18) CPU and GPU execution timings. For each ESN configuration, 20 timing measurements were taken. Plotted here are the mean and standard deviation of the times measured. See Footnote 3 on the preceding page.

Table 4.3: The Echo State Network, CPU and GPU timings – GPU speedup. This was calculated using the mean and standard deviation timings shown in Figure 4.1 as per Section 2.5.2. See Footnote 3 on the preceding page.

Reservoir Size	ESN Execution: ESN Speed-up	
	Double Precision	Single Precision
16	**0.2130 ± 0.1314**	**0.2107 ± 0.1048**
32	0.2368 ± 0.1483	0.2486 ± 0.1076
64	0.2602 ± 0.0600	0.2227 ± 0.1153
128	0.2944 ± 0.0416	0.2944 ± 0.0392
256	0.3499 ± 0.1034	0.3590 ± 0.0891
512	0.6151 ± 0.1308	0.5498 ± 0.1500
1024	2.0243 ± 0.0314	1.4407 ± 0.1164
2048	**5.9923 ± 0.0563**	**4.9652 ± 0.0893**

In both the CPU and GPU cases, the Echo State Network execution time increases with reservoir size. This is not surprising, as the reservoir size determines both the number of rows and columns in the matrix **W** (Equation 2.1). The GPU implementation gives a speed-up at reservoir sizes of 1,024 and 2,048 (Table 4.3). The largest speed-up, 5.9923, is observed for a reservoir size of 2,048 in double precision. The largest slow-down is 0.2107 at a reservoir size of 16 in single precision.

For small ESNs, it is likely that host-GPU memory transfers dominate ESN calculation time. Also, it is probable that the GPU is not fully occupied, and therefore not performing at full capacity or efficiency. The slower clock speed of the GPU will also contribute to a slower than CPU execution time. As the ESN reservoir sizes become larger, it is likely that the occupancy of the GPU improves, and the dominance of host-GPU memory transfers decreases. The CPU, running 4 cores and 8 threads, reaches its computational capacity earlier than the GPU, which has 480 cores and 23,040 threads (see Table 4.1). GPU thread occupancy and the impact of memory transfers is yet to be measured.

4.2.5 Tikhonov Regularisation Speed

The results of the Tikhonov regularisation (TR) execution timing measurements are given in Figures 4.2 and 4.3 and in Table 4.4. Figure 4.2 gives the timing results in the single precision case. Figure 4.3 plots the double precision timings. Table 4.4 presents the mean GPU speedup. Due to space restrictions, only a selection of the speed-ups are presented in Table 4.4. The speed-ups selected are for reservoir sizes $N = 16$ and $N = 2,048$. These represent the two extremes of GPU performance in this experiment. (See Footnote 3 on page 70.)

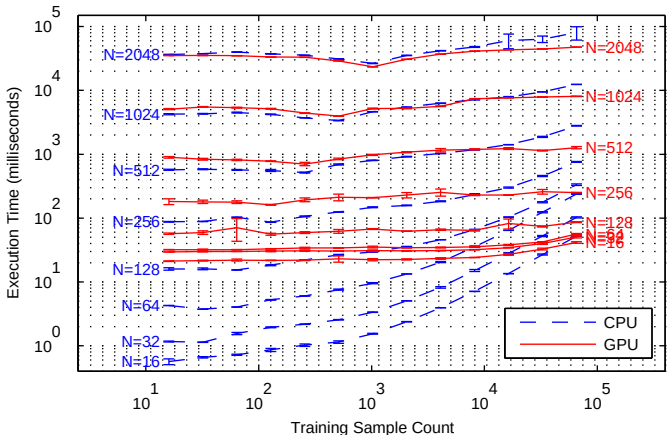

Figure 4.2: The Tikhonov regularisation, mean (Equation 2.17) and standard deviation (Equation 2.18) CPU and GPU *single precision* execution timings. Each curve plots the results for a single reservoir size on either the CPU or the GPU. For each TR configuration, 20 timing measurements were taken. Plotted here are the mean and standard deviation of the times measured. See Footnote 3 on page 70.

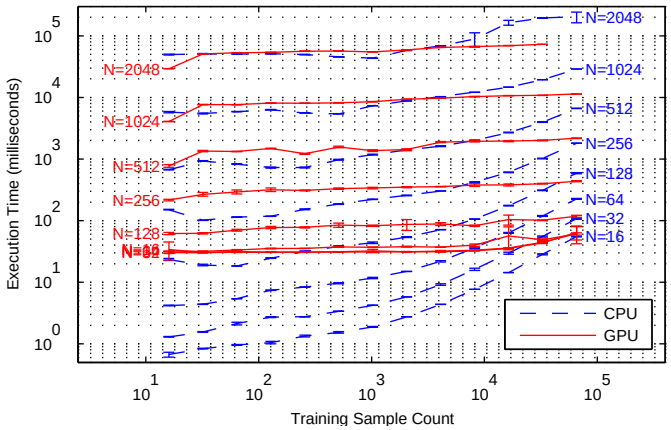

Figure 4.3: The Tikhonov regularisation, mean (Equation 2.17) and standard deviation (Equation 2.18) CPU and GPU *double precision* execution timings. Each curve plots the results for a single reservoir size on either the CPU or the GPU. For each TR configuration, 20 timing measurements were taken. Plotted here are the mean and standard deviation of the times measured. See Footnote 3 on page 70.

As with the Echo State Network execution times, the Tikhonov regularisation execution times increase with reservoir size, N. Additionally, TR execution time increases as the number of training samples ($T - T_0$) increases. This is expected, as the reservoir size effects the number of rows in $\mathbf{X}$, and the number of training samples determines the number of columns in $\mathbf{X}$ (see Equations 2.13 and 2.10).

In the case where the reservoir size was $N = 16$ one speed-up of 1.2571 occurred where the number of training samples was $T - T_0 = 65,536$ in single precision, all other measures gave a slow-down. The largest slow-down, 0.0197, was observed at $T - T_0 = 16$ in double precision. It should be noted that several of the calculated speed-ups in this set have accumulated standard deviations that are larger than the mean, which implies that the variability of measurements at these points is too high to give an accurate measure. Future experiments should address this issue (See Section 5.1.9).

In the $N = 2,048$ case, speed-ups were observed at more than half of the measurement points, excluding the double precision, $T - T_0 = \{64, \ldots, 2048\}$ case. The greatest speed-up, 2.6813, was observed at $T - T_0 = 32,768$ in double precision. The largest single precision speed-up, 1.6864, was observed at $T - T_0 = 65,536$. The greatest slow-down, 0.7910, was observed at $T - T_0 = 1,024$ in double precision. It should be noted that in the $N = 2,048$, double precision case, the measurement at $T - T_0 = 65,536$ could not be taken, as the GPU had reached its global memory limits.

The marginal speed-ups observed may be partly attributed to host-GPU memory transfers that take place. The current implementation uses Magma's SVD imple-

Table 4.4: The Tikhonov regularisation, CPU and GPU timings – GPU speedup. This was calculated using the mean and standard deviation timings shown in Figures 4.2 & 4.2 as described in Section 2.5.2. See Footnote 3 on page 70.

Training samples ($T - T_0$)	TR Execution: GPU Speed-up			
	$N = 2048$		$N = 16$	
	Double Precision	Single Precision	Double Precision	Single Precision
16	1.6961 ± 0.0156	$\mathbf{1.0357 \pm 0.0107}$	$\mathbf{0.0197 \pm 0.3367}$	$\mathbf{0.0266 \pm 0.1029}$
32	1.0022 ± 0.0187	1.0675 ± 0.0062	0.0273 ± 0.0158	0.0306 ± 0.0270
64	0.9499 ± 0.0067	1.1372 ± 0.0079	0.0305 ± 0.0364	0.0329 ± 0.0423
128	0.9498 ± 0.0045	1.1199 ± 0.0048	0.0340 ± 0.0456	0.0397 ± 0.0566
256	0.8814 ± 0.0066	1.0737 ± 0.0054	0.0435 ± 0.0342	0.0467 ± 0.0458
512	0.8012 ± 0.0076	1.0735 ± 0.0077	0.0495 ± 0.0303	0.0498 ± 0.1220
1024	$\mathbf{0.7910 \pm 0.0076}$	1.1291 ± 0.0067	0.0608 ± 0.0196	0.0693 ± 0.0406
2048	0.9824 ± 0.0091	1.1499 ± 0.0059	0.0879 ± 0.0095	0.1059 ± 0.0201
4096	1.0605 ± 0.0075	1.1310 ± 0.0061	0.1398 ± 0.0149	0.1690 ± 0.0277
8192	1.3258 ± 0.2617	1.1621 ± 0.0158	0.2366 ± 0.0024	0.2950 ± 0.0040
16384	2.3569 ± 0.0929	1.3997 ± 0.2584	0.4067 ± 0.0031	0.5010 ± 0.0325
32768	$\mathbf{2.6813 \pm 0.0118}$	1.4287 ± 0.1147	0.6557 ± 0.0307	0.8196 ± 0.0361
65536	–	$\mathbf{1.6864 \pm 0.2362}$	$\mathbf{0.8561 \pm 0.2424}$	$\mathbf{1.2571 \pm 0.0289}$

mentation. The Magma SVD requires inputs from, and returns outputs to host memory; whereas the TR implementation generates SVD inputs and processes SVD outputs on the GPU. This necessitates additional host-GPU memory transfers. While it is likely that these transfers impact the GPU TR execution time, the actual impact of these transfers is yet to be assessed.

4.3 Predictive Performance

The behaviour of the Echo State Network as a multi-time-step predictor was assessed in several experiments that are presented here. The experiments aimed to show how reservoir size and output feedback (the $\mathbf{W}^{\text{ofb}}$ term in Equation 2.1) affect the learning capabilities of the Echo State Network. In each experiment, the ESN was trained to accept a time-series, and to predict the value of this time-series multiple samples ahead. For each time-series, the reservoir size and presence of output feedback were varied, and the accuracy of the ESN was measured. Three time-series were tested. The first, a sinusoid; the second, a Mackey-Glass time-series; and the third, a multiple superimposed oscillator (MSO). The accuracy of each ESN configuration is measured using the normalised root mean squared error (NRMSE) defined in Equation 2.21. Following are the problem statements, the experimental configuration, and finally the results.

4.3.1 Problem Statements

This section describes the three prediction problems attempted in these experiments. Each problem describes a discrete input time-series that drives the Echo State Network, and a discrete output time-series that the ESN must generate. The output time-series is the input time-series advanced by some number of samples. The ESN

is assessed on its ability to generate the output time-series when presented with the input time-series.

To clarify, for each problem, the Echo State Network was presented with the discrete input time-series $u[t]$, and was trained to produce the expected discrete output time-series $x[t]$, where

$$x[t] = u[t+n].$$

Here, n is the lead sample count and is $n > 0$, thus defining the amount that the output advances the input. The ESN is trained over the sample set $T_0 \leq t \leq T$.

The accuracy of the trained ESN was tested by presenting it with the input time-series $u[t]$, and observing its output, $\hat{x}[t]$ (the ESN's estimate of $x[t]$). This was done for some testing sample set $T+1 < t < S$. The accuracy of the the ESN was thus calculated as the NRMSE error (Equation 2.21) between the estimated output $\hat{x}[t]$ and the expected output $x[t]$ for $T < t < S$.

The following sections describe the three problems in terms of input equation $u[t]$, and expected output equation $x[t]$, where $\triangle T$ is the sample period. The values used for $\triangle T$ and the lead sample count, n, are given with the experimental parameters in Section 4.3.3.

4.3.1.1 The Sinusoid Problem

A simple demonstration of an Echo State Network's behaviour is sinusoid generation, as described in [13]. Here the sinusoid is used to drive the ESN, and the ESN must predict the value of the sinusoid n samples ahead. For the given input

$$u[t] = sin(t \triangle T),$$

the Echo State Network shall generate the output

$$x\left[t\right] = sin\left(\left(t+n\right)\triangle T\right).$$

4.3.1.2 The Mackey-Glass Problem

In [13] and [5], the Echo State Network was shown to perform well when generating a Mackey-Glass time-series [58]. The solution to this time-delay differential equation can produce complex dynamics and chaos, and is therefore a difficult problem.

In this problem, a discretised version of the Mackey-Glass time-series was used as follows. For the given input, a solution to the discrete differential equation

$$u'\left[t\right] = \beta\frac{u\left[t-d\right]}{\left(1+u\left[t-d\right]^{m}\right)} - \gamma u\left[t\right], \tag{4.1}$$

the Echo State Network shall generate the solution to the discrete differential equation

$$x'\left[t\right] = \beta\frac{x\left[t-d+n\right]}{\left(1+x\left[t-d+n\right]\right)} - \gamma x\left[t+n\right].$$

Where $\beta = 2$, $m = 9.65$, $\gamma = 1$, and $d = \left\lfloor\frac{2}{\triangle T}\right\rfloor$.

Here, d is the discrete-time delay, usually expressed as τ in the continuous-time case. Also note that the variable given here as $m = 9.65$, is usually labelled as n. The label m has been used to avoid confusion with the lead sample count, n, already defined in this section.

The time-series used in these experiments was generated numerically. First, the discrete differential equation, Equation 4.1, was generated. Then, a numerical solution for $u\left[t\right]$ was found using the fourth-order Runge-Kutta method [59] with initial

conditions of $u[t] = 0.5$ for $-d \leq t \leq 0$.

4.3.1.3 The Multiple Superimposed Oscillators (MSO) Problem

The multiple superimposed oscillators (MSO) problem is described in [60] as a difficult problem to solve using the "classical" Echo State Network approach, first described in [3]. Despite this, solutions have been found [60, 55].

In [60], Wierstra et al. cite a 2004 lecture[4] in which Jaeger states that the MSO problem can be difficult to solve. The reason given is that, when the wavelength of the MSO is long, a large number of samples are required to capture an entire period of the waveform. Also, as the dynamics of the ESN's reservoir are coupled, it is difficult for the reservoir to represent the two independent oscillators described in the equations. Despite this, Wierstra [60] and Steil [55] have found solutions to the problem. Steil goes so far as to say that the problem is "too simple", to be used as a bench-marking problem [55].

For this work the MSO problem is defined as follows. For the given input

$$u[t] = sin(0.2t \triangle T) + sin(0.311t \triangle T),$$

the Echo State Network shall generate the output

$$x(t) = sin(0.2(t+n) \triangle T) + sin(0.311(t+n) \triangle T).$$

4.3.2 Performance Assessment

For each experiment, random Echo State Networks were generated over a range of reservoir sizes. Each ESN was trained, then its performance was assessed based

[4]The slides referenced in [60] are no longer available.

on its ability to reproduce a test data set that the network had not yet seen. The training and test data sets were taken from contiguous sections of the time-series in question. The training data first, then the test data.

4.3.2.1 Initial Conditions & Execution

When assessing the performance of a trained Echo State Network, the ESN was configured with a null initial state,

$$\mathbf{x}(0) = \mathbf{0}.$$

The ESN was then run, over the entire training and test sequence, in a contiguous manner.

4.3.2.2 Numerical Input & Operations

For this experiment, the same input and training values were used in both CPU and GPU cases. However, the exact structures of the Echo State Networks used differ between both implementations. This is because the ESNs were generated pseudo-randomly using different pseudo-random number generators. Thus, the numerical operations performed in the CPU and GPU cases will not be identical, and could perhaps hinder their comparability. This could be addressed in future work (see Section 5.1.8).

4.3.3 Experimental Parameters

In [3, 13], it is stated that the parameters chosen for an Echo State Network are relative, if not unique to a problem. Although the correctness of this statement is not

in doubt, all three prediction experiments have a set of largely common parameters. This was done to increase the comparability of the results across the three problems. A complete list of the experimental parameters is given in Table 4.5.

Of the experimental parameters, the maximum reservoir size and the reservoir connectivity were chosen based on [5]. Unlike the previous experimental parameters (Section 4.2.3.5), the Tikhonov regularisation factor in this experiment was set to $\lambda = 0$. This was done to remove the effect of choosing a "non-optimal" TR factor (see Section 2.2.5.3). The spectral radius of the reservoir, $\rho(\mathbf{W})$, was chosen arbitrarily, but within the bounds described in Section 2.2.5. Unlike the previous experiment, the Echo State Networks were tested both with and without output feedback (the $\mathbf{W}^{\text{ofb}}$ term in Equation 2.1). The sample period, $\triangle T$ was chosen within the bounds $[0.05, 0.5]$, which is given in [55] as an appropriate range for solving the MSO problem. Note that the sample period and ESN time-step are identical, as recommended in [55]. Finally, the lead time was chosen arbitrarily as $\delta = \pi$. Thus, the lead sample count was calculated as $n = \left\lfloor \frac{\pi}{\triangle T} \right\rfloor = 31$.

Table 4.5: Time-series prediction – the Echo State Network, and ESN training configuration parameters. Note that the non-zero values in the weight matrices are all drawn from the uniform distribution.

Parameter	Value
Sample period, $\triangle T$. The amount of time between each time-step.	0.1
Lead time, δ.	π
Lead sample count, $n = \left\lfloor \frac{\delta}{\triangle T} \right\rfloor$.	31
Reservoir sizes, N (i.e. the sizes of $\mathbf{x}$, Eq. 2.1).	$\{25, 50, \ldots, 975, 1000\}$
ESNs measured per reservoir size.	40
Reservoir connectivity (proportion of non-zero values in $\mathbf{W}$, Eq. 2.1).	0.01
Spectral radius ($\rho(\mathbf{W})$, Eq. 2.3).	0.9
Presence of output feedback (the $\mathbf{W}^{\text{ofb}}$ term, Eq. 2.1)	{present, absent}
$\mathbf{W}^{\text{in}}$ weight range (Eq. 2.1).	$(-1, 1]$
$\mathbf{W}$ weight range (Eq. 2.1).	$(-1, 1]$
$\mathbf{W}^{\text{ofb}}$ weight range (Eq. 2.1).	$(-1, 1]$
Tikhonov regularisation parameter (λ, Eq. 2.13).	0
Number of test and training vectors available, S.	10,000
Number of samples discarded during training, T_0, to reduce the impact of initial transients (see Sect. Section 2.2.5.1).	2,000
Number of samples used in Tikhonov regularisation, $T - T_0$ (Eqs. 2.10 & 2.13).	4,969
Number of samples used to test ESN performance.	2,969

4.3.4 Results

For each of the prediction experiments, the collected results are presented in three parts. First, the *median* error performance is analysed. Here, the errors from the three problems are compared. Second, cases when feedback is applied and withheld are compared for each of the three problems. Finally, the outputs from the *best*

performing Echo State Networks are given. Used throughout this section is the *normalised root mean squared error* (or *NRMSE*) as defined in Equation 2.21. The errors presented in this section are either NRMS errors, or quartiles thereof. During the following discussion, the problems in Section 4.3.1 are referred to as the *sinusoid problem* (Section 4.3.1.1), the *Mackey-Glass problem* (Section 4.3.1.2), and the *MSO problem* (Section 4.3.1.3).

4.3.4.1 Median Error Performance

The median error performance of the Echo State Network is given in Figures 4.4 and 4.5, and in Table 4.6. The figures plot *median* NRMS error against reservoir size, and the table lists the *minimum* median errors observed. Figure 4.4 plots specifically the median error curves of ESNs *with* output feedback (the $\mathbf{W}^{\text{ofb}}$ term in Equation 2.1). One median NRMS error curve is plotted for each of the three problems given in Section 4.3.1. Similarly, Figure 4.5 plots error curves of ESNs *without* output feedback. The following paragraphs describe and discuss the results presented in Figures 4.4 and 4.5, and in Table 4.6.

With Output Feedback. Figure 4.4 gives the median NRMS error curves for all three problems, when solved by an ESN *with* output feedback. All three error curves display a clear descent to some local minimum, then a gradual ascent. The sinusoid problem reaches an observed minimum median error at a reservoir size of 850, the Mackey-Glass and MSO curves reach their both reach minima at a reservoir size of 775. Unlike the other curves, the sinusoid displays a sudden drop at a reservoir size of approximately 700, thus yielding a significantly lower minimum median error.

The initial descent of each curve indicates an improvement in ESN suitability as the reservoir size increases. This improvement appears to reach a best case, where a

minimum error is observed. Following this minimum, the error gradually increases, which is likely due to overfitting. The sudden drop to a minimum error displayed by the sinusoidal curve indicates a dramatic change of ESN suitability, as though the ESN has crossed some bound. This is undoubtedly related to the dynamics of the reservoir, and would be is an interesting topic of study for future work (see Section 5.1.10).

Without Output Feedback. Figure 4.5 plots the median NRMS error curves for the three problems, when solved by an ESN *without* output feedback. The Mackey-Glass error curve displays a clear descent to a minimum median error at a reservoir size of 475. This is followed be a clear ascent in median error. The sinusoid and MSO curves behave differently. They first ascent to some local maximum, then descend to a local minimum. This indicates the presence of two local error minima. For the sinusoid curve, the observed minimum median error occurs at the *second* local minimum, at a reservoir size of 950. The MSO curve has its observed minimum at the *first* local minimum, at a reservoir size of 25. As in the with feedback case, the sinusoid error curve displays a sudden drop, although not as large as in the with feedback case. This behaviour warrants further investigation (see Section 5.1.10).

The Lowest Median NRMS Error. Table 4.6 presents the lowest median NRMS errors observed for each of the problems. In both the with- and without-feedback cases, the sinusoid problem yields a lower error than the Mackey-Glass problem, and the Mackey-Glass problem a lower error than the MSO problem. When feedback is present, the minimum sinusoid error is significantly lower than the minimum errors of the other problems.

When comparing the with- and without-feedback cases, the sinusoid problem has a significantly lower median error in the presence of feedback. The Mackey-Glass and

MSO problems, on the other hand, display a lower error when feedback is absent.

These results indicate that the sinusoid problem can be solved to a significantly higher degree of accuracy than the other problems when feedback is used. The results also imply that, for the Mackey-Glass and MSO problems, it is better to use an ESN without feedback. This may be because the output feedback introduces instabilities problems that are difficult to compensate for. Further work on the impact of output feedback on stability is warranted (see Section 5.1.12).

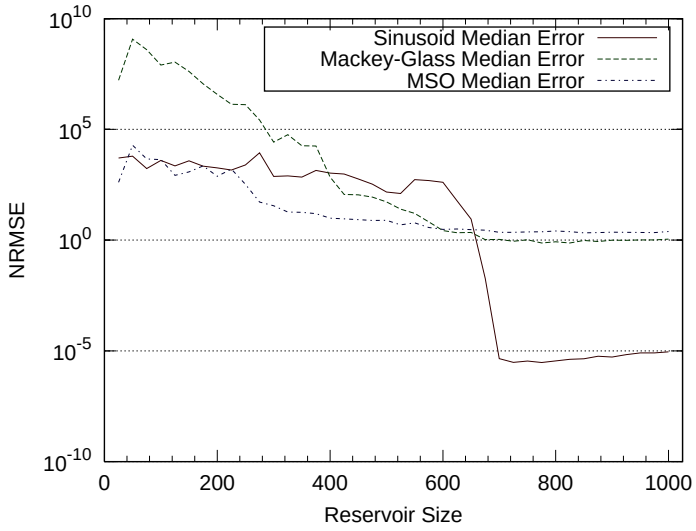

Figure 4.4: Double precision prediction problems *with* output feedback – *median* measured NRMS errors per reservoir size (see Section 2.5.4).

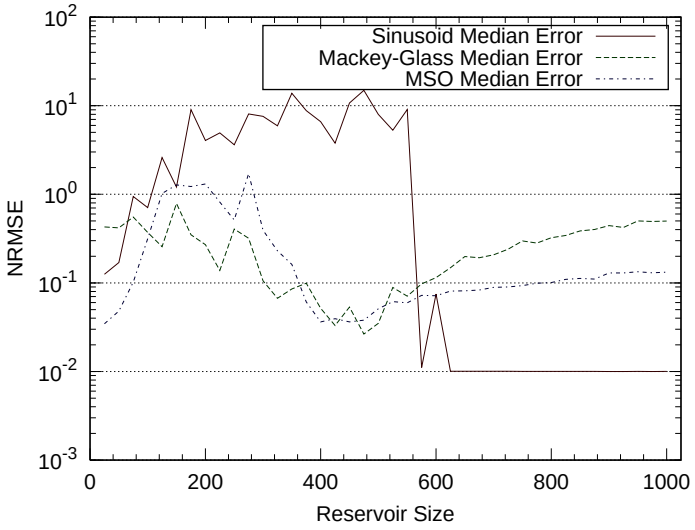

Figure 4.5: Double precision prediction problems *without* output feedback – *median* measured NRMS errors per reservoir size (see Section 2.5.4).

Table 4.6: The lowest median NRMS errors observed in each of the three problems.

Problem	Feedback present		Feedback absent	
	Min. median NRMSE	Reservoir size	Min. median NRMSE	Reservoir size
Sinusoid	2.9825×10^{-6}	775	0.0099926	950
Mackey-Glass	0.74300	775	0.026425	475
MSO	2.1333	850	0.034581	25

4.3.4.2 With- & Without-Feedback Performance

The performance measures presented in this section are similar to those given in Section 4.3.4.1. However, attention is given rather to the difference between the with- and without-feedback cases. The spreads of the errors are also given.

The performance of the sinusoid, Mackey-Glass, and multiple superimposed oscillator problems are plotted in Figures 4.6, 4.7, and 4.8 respectively. Each plot presents the results of one problem in four parts. The plots each present two error curves that describe the performance of the with- and without-feedback ESNs. Each plot also presents two points which indicate the lowest observed errors of ESNs both with and without feedback. The curves plot the reservoir size against the median, lower-quartile, and upper-quartile normalised root mean squared error (as defined in Section 2.5.4).

The following paragraphs describe the results given in Figures 4.6, 4.7, and 4.8. This is followed by a discussion of the results. The points of lowest error are expanded in Section 4.3.4.3.

The Sinusoid Problem. Figure 4.6 plots NRMS error data for the sinusoid problem. Here, the *without*-feedback curve is lower for smaller reservoir sizes, and higher for larger reservoir sizes. The without-feedback curve also shows a smaller range. The absolute minimum is significantly lower in the with-feedback case, the reservoir size, however, is much larger.

The Mackey-Glass Problem. Figure 4.7 presents the NRMS error data for the Mackey-Glass problem. Here, the *without*-feedback curve is consistently lower than the with-feedback curve, and also has a smaller range. The *without*-feedback curve is also much lower for smaller reservoir sizes. The absolute minimum is, however, significantly lower in the *with*-feedback case. It is also significantly further from the median, and with a notably larger reservoir size.

The MSO Problem. Figure 4.8 plots the NRMS error data for the MSO problem. Here, the *without*-feedback curve is consistently lower than the with-feedback curve,

and has a smaller range. The absolute minimum is significantly lower in the *without*-feedback case, and at a much lower reservoir size.

Discussion. The NRMS error curves described above demonstrate clear differences when using ESNs with, and ESNs without output feedback. In all cases, the range of the error curves in the without-feedback case were smaller than in the with-feedback case. In the sinusoid and Mackey-Glass problems, the curves and minimum points suggest that using output feedback has potential to yield lower absolute errors, but with higher average errors. In the MSO problem, output feedback appears to be more of a hindrance than a help.

The above without-feedback NRMS error curves all display a smaller range than in the with-feedback case. Furthermore, at smaller reservoir sizes, the ESN's without feedback produced lower errors. This could be attributed to the relative instability of ESN's with-feedback [2]. It is possible that, in some cases, output feedback introduces an instability that the trained output weights can not compensate for at lower reservoir sizes. In other words, a larger reservoir size may be required to compensate for instabilities introduced by the output feedback. Thus, the with-feedback ESN's may have produced a wider range of errors per reservoir size, and therefore a higher average error in the above curves.

Although the ESN's without feedback produced error curves with a smaller range, in two problems, the lowest absolute errors were produced by ESN's with feedback. The sinusoid and Mackey-Glass problems both gave lowest absolute errors when solved by ESN's with feedback. This is in contrast with the MSO problem. Here, an ESN without feedback produced the lowest absolute error. A study of the literature on the the effects of output feedback, and when best to use it, is proposed for future work (see Section 5.1.12).

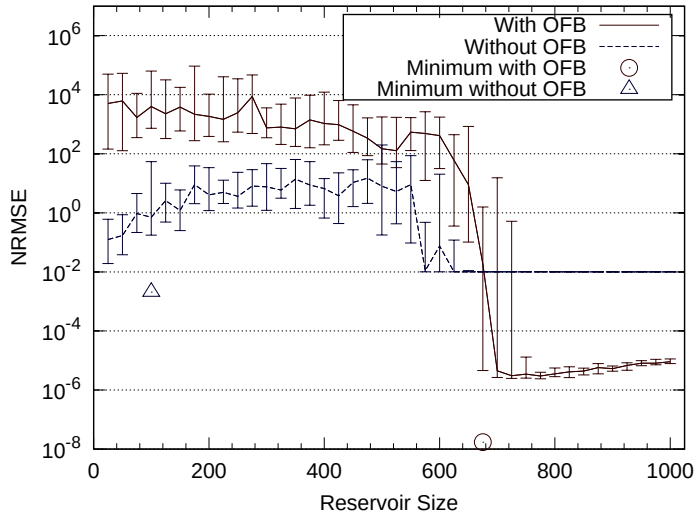

Figure 4.6: Double precision sinusoid prediction – measured NRMS error per reservoir size. Plotted are the median, lower, and upper quartile errors (see Section 2.5.4).

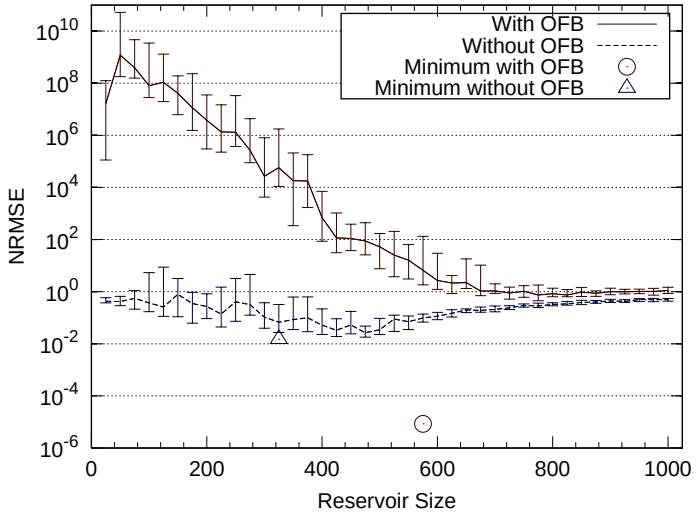

Figure 4.7: Double precision Mackey-Glass time series prediction – measured NRMS error per reservoir size. Plotted are the median, lower, and upper quartile errors (see Section 2.5.4).

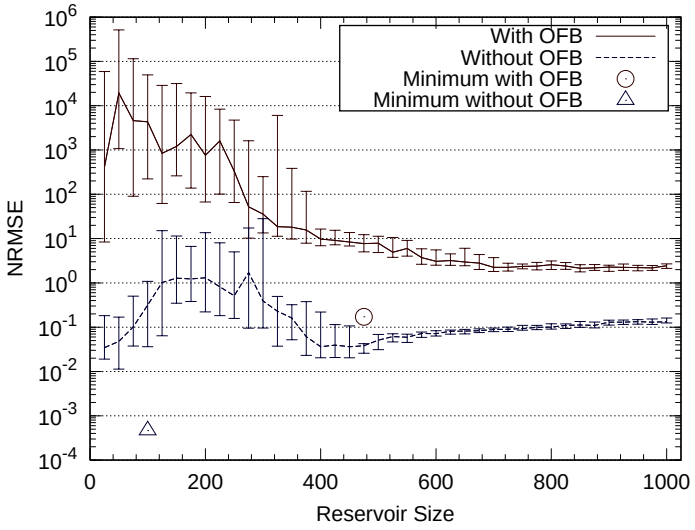

Figure 4.8: Double precision MSO prediction – measured NRMS error per reservoir size. Plotted are the median, lower, and upper quartile errors (see Section 2.5.4).

4.3.4.3 Lowest NRMS Error ESN's

The outputs of the Echo State Networks that gave the lowest error are presented in this section. The outputs are presented in Table 4.7, and in Figures 4.9, 4.10, 4.11, 4.12, 4.13, and 4.14. For each of the three problems, two plots are given. One presents the outputs of the lowest error ESN with feedback, and the other the outputs of the lowest error ESN without feedback.

The best Echo State Network performance for each of the three problems is given in Table 4.7. Here, we observe that an ESN with feedback solved the sinusoid problem to a significantly higher degree of accuracy than the Mackey-Glass problem. The Mackey-Glass problem was, in-turn, solved to a higher degree of accuracy than the MSO problem. This is in contrast to the without-feedback case. An ESN without feedback solved the MSO problem to a significantly higher degree of accuracy than

the sinusoid problem. The sinusoid problem was, in-turn, solved to a higher degree of accuracy than the Mackey-Glass problem.

One can also compare the with- and without-feedback cases of each problem. Here, the sinusoid problem is significantly better solved with feedback, but with a significantly larger reservoir size. The Mackey-Glass problem, on the other hand, is solved to a similar accuracy by ESN's both with and without feedback. The without-feedback reservoir size, however, is significantly larger. The MSO problem is also in contrast, as the best performing ESN was without feedback, and with a smaller reservoir size. Interestingly, in all problems, the absolute minimum achieved by ESNs without-feedback is at a significantly lower reservoir size than in the with-feedback case. Which, when using the results from Section 2.5.2, should result in a faster calculation time. Thus, for these problems, the ESN designer is faced with a trade-off between accuracy and speed.

Table 4.7: Lowest observed ESN NRMS errors.

	Feedback present		Feedback absent	
Problem	Min. NRMSE	Reservoir size	Min NRMSE	Reservoir size
Sinusoid	1.7233×10^{-8}	675	0.0020683	100
Mackey-Glass	0.018256	625	0.014925	325
MSO	0.17102	475	4.6421×10^{-4}	100

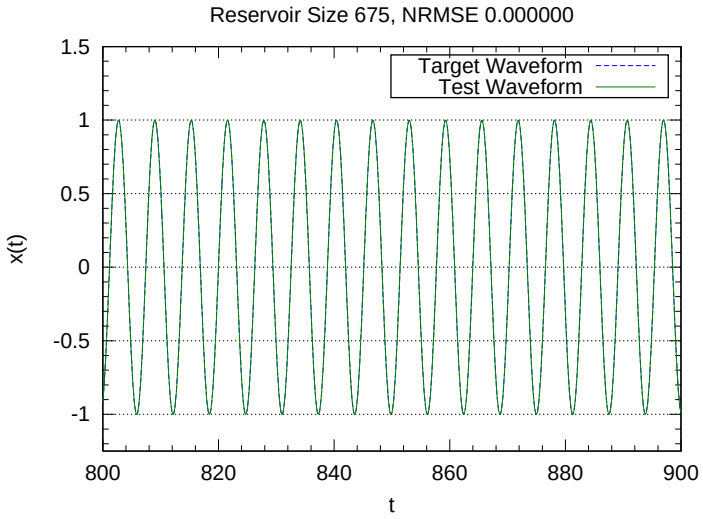

Figure 4.9: Double precision sinusoid prediction – a portion of the lowest error *test* waveform. This occurred at a reservoir size of 675, with an NRMS error of 1.7233×10^{-8}.

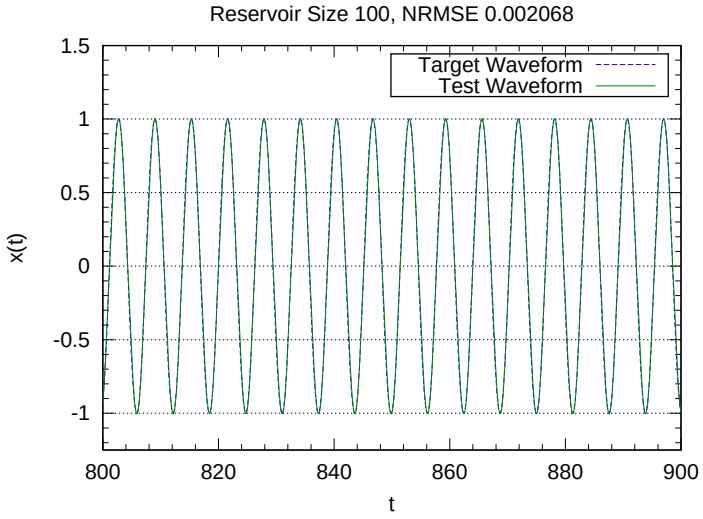

Figure 4.10: Double precision sinusoid prediction – a portion of the lowest error *test* waveform. This occurred at a reservoir size of 100, with an NRMS error of 0.0020683.

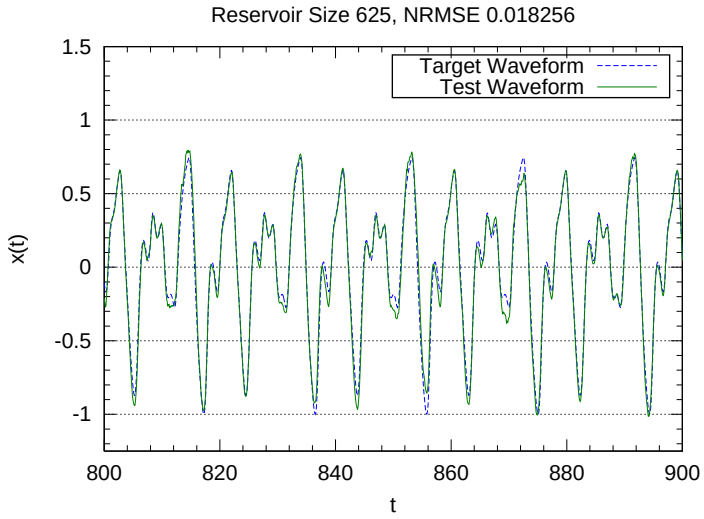

Figure 4.11: Double precision Mackey-Glass time series prediction – a portion of the lowest error *test* waveform. This occurred at a reservoir size of 575, with an NRMS error of 8.4726×10^{-6}.

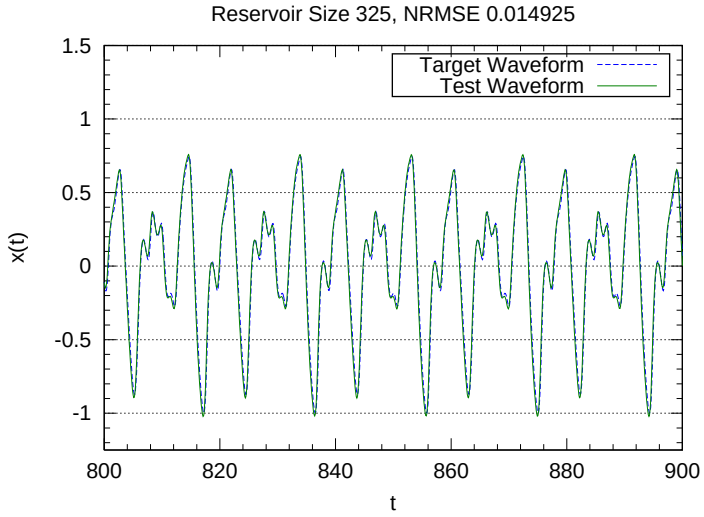

Figure 4.12: Double precision Mackey-Glass time series prediction – a portion of the lowest error *test* waveform. This occurred at a reservoir size of 225, with an NRMS error of 0.0086278.

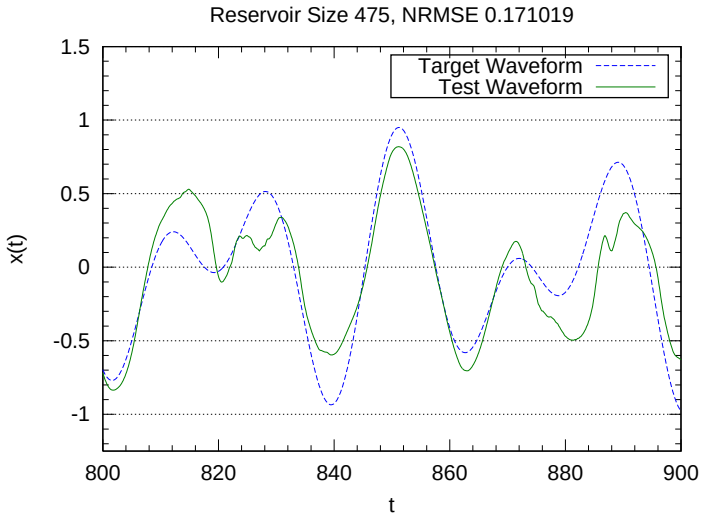

Figure 4.13: Double precision MSO prediction – a portion of the lowest error *test* waveform. This occurred at a reservoir size of 850, with an NRMS error of 0.037355.

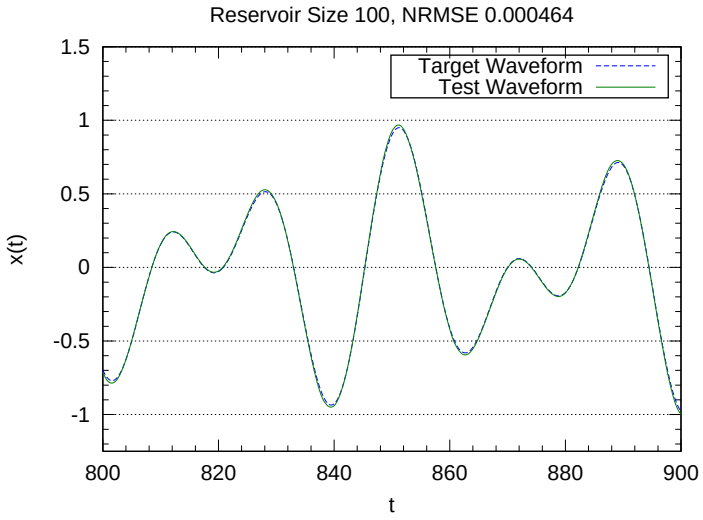

Figure 4.14: Double precision MSO prediction – a portion of the lowest error *test* waveform. This occurred at a reservoir size of 175, with an NRMS error of 0.012298.

5 Conclusion

The Echo State Network, a form of recurrent artificial neural network, was implemented for the GPU. The form of ESN built was that described in the original work, [3]. An offline training method based on Tikhonov regularisation [15, 2] was also implemented. The implementation targeted the Nvidia Cuda platform, and used a combination of bespoke kernels, and existing Cuda BLAS [37], Sparse [24], random number [25], and LAPACK [42] libraries. To ensure fast memory transfers between the host and the GPU, pinned and write-combining memory [37] were used. Cuda streams were used to implement concurrent execution patterns, including a double-buffering system [39] for the calculation of reservoir state and ESN output equations. UML sequence diagrams illustrating the concurrent behaviour of the ESN are given in Figures 3.2, 3.3 and 3.4.

The behaviour of this GPU Echo State Network implementation was examined. First to assess its speed against a CPU implementation, and second to assess its behaviour with several prediction problems. These experiments were conducted on a single test computer with a Intel i7-980 CPU and an Nvidia GTX480. Both representative of high-end commodity hardware in their respective classes.

To assess the speed of the GPU ESN, two experiments were devised. These targeted the two key components of ESN execution, the calculation of ESN output, and calculating the ESN's output weights via Tikhonov regularisation. These were compared

with a CPU implementation. This CPU implementation used GNU-Octave [54] which interfaced with the reference BLAS [40] and LAPACK [41] implementations via the automatically tuned linear algebra system [57].

The first speed experiment captured the execution time of the GPU and CPU ESN output calculations over a range of reservoir sizes in both double and single precision floating point. The remaining variables were fixed. The timing results are presented in Figure 4.1 and their corresponding speed up calculations are compiled in Table 4.3. The speed-up ranged from 0.2107 to 5.9923, the former was obtained with a reservoir size of 16 in single precision, the latter with a reservoir size of 2048. In both single and double precision cases, a GPU speed-up was observed for reservoir sizes over 512.

The second speed experiment captured the execution time of the Tikhonov regularisation algorithm over a range of reservoir sizes and state-history sizes. This was performed in both double and single precision. The timing results are presented in Figures 4.2 and 4.3. The calculated speed-ups are listed in Table 4.4. These ranged from 0.0197 to 2.6813. In the single precision case, with a reservoir size of 2048, the GPU gives a slight speed-up for every configuration. In all other cases, speed-ups are observed only for the largest state-history sizes. Memory limitations on the GPU meant that a measurement could not be made in the double precision case, for a reservoir size of 2048, and a training sample size of 65,536.

To further assess the behaviour of this GPU Echo State Network implementation, several learning experiments were devised. These tested the behaviour of the ESN when predicting three studied time-series [3, 55]. A sinusoidal time-series, a Mackey-Glass time-series [58], and a multiple superimposed oscillator (MSO) [60, 55]. In each experiment, ESN's were driven by the time-series, and trained to predict the value of the time-series multiple-samples ahead. The trained ESN's were evaluated

by driving them with a previously unseen portion of the time-series, capturing their outputted predictions, and comparing their estimates with the expected output. The comparison was made using a normalised root mean squared error measure. These tests were performed in double precision over a range of reservoir sizes, and used ESN's both with and without output feedback. The remaining ESN parameters were fixed. Thus, six cases were studied – three time-series, each solved by ESN's with and without output feedback. Multiple ESN's were measured for each reservoir size.

The results of these experiments are given in Tables 4.6, and 4.7, and Figures 4.6, 4.7 and 4.8. All cases demonstrated a relationship between reservoir size and median error performance. A descending median error implies increasing ESN suitability. An ascending median error may imply overfitting, although in two of the six cases, two local error minima and one local maximum were observed. For each time-series, the ESN's without feedback produced median error curves with smaller ranges than ESN's with feedback. ESN's with feedback produced higher median errors at smaller reservoir sizes than the ESN's without feedback. However, the ESN's with feedback produced the lowest absolute errors in the sinusoid and Mackey-Glass cases. In contrast, for the MSO case, output feedback produced no observable benefits. In all three problems, the absolute minimum error in the with-feedback case, was produced at a significantly higher reservoir size than in the without-feedback case.

The results of the speed experiments demonstrate the benefits that GPUs can provide to Echo State Network designers. However, the GPU does not provide benefits in all cases. In other words, it is not a "golden hammer"[1]. When comparing the Intel i7-980 CPU and an Nvidia GTX480, the GPU ESN was shown to run

[1]This is the idea that one tool can solve all of one's problems. If all of one's problems are viewed as a "nail", then the solution to of all of one's problems can be provided by a single "golden hammer".

faster than the CPU ESN for reservoir sizes over 512. Smaller reservoir sizes gave a slow-down. The largest observed speed-up was 6. Extending the experiment to larger reservoir sizes would likely yield speed-ups greater than this. Although, these would be eventually limited by GPU memory size, as observed in the Tikhonov regularisation problem. Given the architectures of the GPU and the CPU, these are not surprising results, however, the point at which the GPU gives a speed-up is of interest. For different hardware, this cross-over point will be different, and the size of both CPU and GPU memory will form an upper bound on the size of the problems that can be solved. Therefore users must take care to run similar tests on their own hardware before concluding that the GPU or CPU is the best choice for their specific problem.

The speed-ups observed when running the GPU ESN were not translated to the Tikhonov regularisation problem. Largely, slow-downs were observed; most speed-ups occurred at the very largest state-history sizes, and approaching the limits of GPU memory in the double precision case. Extending the experiment to larger reservoir and / or training sample sizes in the single precision case would likely yield larger speed-ups. There is potential to improve upon this implementation (see Section 5.1), and larger speed-ups may be achieved if deficiencies are found in the current implementation. Based on these results, this implementation of GPU based Tikhonov regularisation can only be recommend when the reservoir size and training sample size are large. Users should, however, consider their hardware. These results will not translate exactly to a different hardware configuration.

The second set of experiments demonstrated a relationship between reservoir size, output feedback, and ESN suitability. In each problem, the ESN's without feedback produced median error curves with a smaller range. This may be attributed to instabilities introduced by output feedback. ESN's with feedback produced higher

median errors at smaller reservoir sizes than the ESN's without feedback. However, the lowest absolute minimum errors were produced, in two problems, by ESN's with feedback at higher reservoir sizes. These results suggest that feedback instabilities can be compensated for by larger reservoirs. Also, using feedback with a larger reservoir may produce a highly accurate ESN, however, this can not be guaranteed. The absolute minimum errors produced by the ESN's with feedback were at significantly higher reservoir sizes than the absolute minimum errors produced by ESN's without feedback. Given the results from the speed experiments, in Section 2.5.2, it is likely that the best performing ESN without feedback would execute faster than the best performing ESN with feedback. Thus, for these problems, the ESN designer faces a trade-off between speed and accuracy.

While these results have provided some insight into the behaviour of a GPU-based Echo State Network, there are many possible improvements and extensions to this work. The final sections discuss these.

5.1 Future Work

This GPU Echo State Network implementation has been demonstrated to behave as expected, and to offer some speed advantages over a CPU implementation. There remains, however, much work that can be done to improve performance, provide more general-case speed comparison results, broaden hardware compatibility, and improve user accessibility and usability. These topics are addressed in the remaining sections.

5.1.1 Profiling & Optimisation

The Nvidia tool-chain provides tools for profiling Cuda applications (see Section 2.3.2). The GPU Echo State Network implementation presented here has not yet undergone profiling assessment. Profiling the implementation may reveal deficiencies in the utilisation of the GPU and memory transfers. The double buffering approach described in Section 2.3.1.5 may not completely "hide" the memory transfers, thus alternative approaches to concurrency may yield performance improvements. Any work that improves GPU utilisation will impact on the results observed in Section 4.2.4.

5.1.2 Sparse Matrix Format

The calculation of the $\mathbf{W}\mathbf{x}\,(n-1)$ term in Equation 2.1 uses a sparse matrix-vector multiplication, and the compressed sparse row (CSR) storage format. The literature shows that a hybrid Ellpack-Itpack and coordinate list implementation (HYB) performs better, except where $\mathbf{W}$ is dense or near dense, and where the number of non-zeros in each row varies greatly (see Section 2.4.3 and [50]). Converting the implementation to use the HYB format is likely to yield improvements over the results given in Section 4.2.4.

5.1.3 Memory Limitations

During the Tikhonov regularisation double precision speed test, the memory limits of the GPU were hit. This occurred for one measurement at a reservoir size of 2048 and a state-history size of 65,536. A GPU with a larger memory would facilitate this measurement. However, it would also be useful for the GPU Echo State Network user to be alerted when a problem is beyond the memory limitations of the GPU.

Further to this, a mechanism to predict a memory problem and to circumvent it would be useful. A prediction mechanism would require knowledge of the available GPU memory, and the amount of memory required for the given calculation. A possible circumvention method would be to store part of the data required for the calculation on the host memory. This would require careful engineering to ensure that the effects of host-GPU memory transfers were minimised. This circumvention method would in-turn be limited by the available host memory; thus, care must be taken not to overburden the host to the point of system failure.

5.1.4 Singular Value Decomposition

The bidiagonalisation step in the Magma SVD routine `magma_*gebrd` makes heavy use of the CPU based `blasf77_*gemv` routine. A higher computation improvement speed may be obtained if this work were migrated to the GPU. Also, modifying the Magma routine to be called with device-side memory may also yield larger speed-ups. Lastly, the Magma libraries used in this implementation may not use the fastest host-side LAPACK and BLAS libraries.

The Magma SVD must be called with the arguments stored in host-side memory, the data is then transferred to the GPU device during the execution of the SVD. For this implementation however, the data required for the SVD operation is already in GPU-memory. Thus, unnecessary transfers between host and device memory are present in this implementation. Modifying the Magma SVD routine to accept arguments stored in GPU memory will likely improve the performance of this implementation.

Similarly, the diagonalisation step is performed entirely on the CPU using the iterative QR routine, `lapackf77_sbdsqr`. If instead this were migrated to the GPU (as per [48, 49]), we may see a reduced computation time. Alternatively, computation

speed may be increased by migrating the LAPACK implementation of Cuppen's divide and conquer algorithm, `*bdsdc`, to the Magma project:

> "`SBDSDC computes the singular value decomposition (SVD) of a real N-by-N (upper or lower) bidiagonal matrix B: B = U * S * VT, using a divide and conquer method, where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and U and VT are orthogonal matrices of left and right singular vectors, respectively.`" [41]

Previous work implementing the Cuppen's divide and conquer algorithm in Magma, has demonstrated a 10 fold improvement over its LAPACK CPU-based counterpart, `*stedc`. Unfortunately, this implementation targets symmetric tridiagonal matrices, and is therefore not suited to this problem. Another approach may be to look into the multiple relatively robust representations (MRRR) method discussed briefly in [43].

As an accompaniment to this work, alternative CPU-based LAPACK and BLAS libraries may be considered for Magma. Magma could be instead compiled against, for example, the Intel math kernel library [61].

5.1.5 Eigenvalue Calculation

This implementation uses a maximum eigenvalue calculation when building a reservoir using the method presented in Section 2.2.3. When performing eigenvalue calculations, it is recommended in [43] to use "canned" eigenvalue packages due to the complexity of this problem. Based on this advice, a canned implementation was employed. However, research into the efficiency of the Magma eigenvalue calculation algorithms used by Magma may yield some improvements. Further to this, Magma can be compiled against alternative CPU-based numerical libraries, for example, the

Intel math kernel library [61], may yield improvements over the reference BLAS and LAPACK libraries used in this implementation.

5.1.6 The CPU Implementation

While the results presented in Section 4.2 do show speed-ups in some cases, it should be noted that the CPU implementation was performed using GNU-Octave, an interpreted language. A compiled or just-in-time compiled CPU-based implementation using, for-example, Intel's math kernel library [61] would make for a fairer comparison. This would then compare an optimised Intel math library on Intel hardware with an optimised Nvidia library on Nvidia hardware.

5.1.7 ESN Input and Output Size

In Section 4.2, the speed experiments did not account for input and output size. This is an important consideration in Echo State Network design. The input and output size used in the experiments was fixed at 16. It could be argued that this gave an advantage to the GPU ESN implementation. As the GPU is optimised for vector and matrix calculations. A future speed performance study should take input and output size into consideration.

5.1.8 ESN Structure During Experiments

In the comparative experiments described in Sections 4.2 and 4.3, exact Echo State Network structures used differ between the CPU and GPU implementations. This is because different pseudo-random number generators were used in the implementations. Similarly, the experiment described in Section 4.2 used pseudo-randomly

generated ESN inputs. These were also generated using different pseudo-random number generators. This may hinder the comparability of the results.

In the GPU implementation, the Curand pseudo-random number generator was used, whereas the CPU implementation uses the Octave pseudo-random number generator. A future implementation could use the same pseudo-random number generator to generate the same ESN structures. The same inputs could also be used in the speed experiments described in Section 4.2.

5.1.9 Variability of Speed-Up Calculations

In Section 4.2, some speed-up calculations had standard deviation values that were larger than the calculated mean. This is due to the propagation of standard deviations from the CPU Echo State Network and the GPU ESN timing measurements. This brings the accuracy of these particular measurements into question. The accuracy of the measurements may be improved by taking more measurements at each reservoir size. For these experiments, 20 measurements were made to obtain each mean and standard deviation speed. Increasing the number of measurements to, say, 40 is likely to reduce the large spread observed for some speed-up calculations.

5.1.10 The Sinusoidal Median Error Curve

In Section 4.3, the sinusoid prediction problem yielded interesting results. Here, the median error produced by Echo State Networks solving this problem displayed a sudden drop. This was observed for ESN's both with and without output feedback. The sudden drop indicates that the ESN's crossed some bound. A more thorough survey of the literature may explain this behaviour.

5.1.11 Multiple Local Minima in Error Curves

In Section 4.3, the sinusoid and multiple superimposed oscillator (MSO) problems yielded interesting results. Specifically when learned by ESN's without feedback. Here, the median errors produced by the ESN's was plotted against reservoir size. Observed in both median error error curves were two distinct minima. The other curves displayed only one. A more thorough survey of the literature may help describe this behaviour.

5.1.12 Output Feedback and ESN Stability

In Section 4.3, the differences observed between Echo State Networks with, and those without output feedback, are of interest. Here, the median errors produced by the ESN's were plotted against the ESN's reservoir size. The ESN's with feedback appeared to produce significantly higher errors at lower reservoir sizes. Which could be attributed to instabilities introduced by the feedback. However, the lowest absolute minimum error was, in two problems, produced by an ESN with output feedback, and at a significantly higher reservoir size than the absolute minimum produced by the ESN without feedback. This suggests that a larger reservoir size can compensate for possible feedback instabilities, and possibly lead to higher accuracy. Some of the already surveyed literature may yield an explanation for this behaviour, and thus indicate potential experimental changes. The possible instabilities observed may be influenced by the spread of the feedback weights, or stability (in the systems-theory sense) of the network, or the absence of output and reservoir regularisation.

The ranges of the output feedback weights were drawn from the uniform distribution with the range $(-1, 1]$, and thus may have been driving some of the reservoir neurons into saturation. Reducing the range to, for example, $(-0.5, 0.5]$ is likely to

eliminate this as a possible cause for network saturation and stability [2]. Furthermore, when building the Echo State Networks in this experiment, the structure of the randomly generated weights was not considered from a systems-theory perspective (see Section 2.2.3). The networks with larger errors are likely to have suffered from a lack of dynamical "richness". The design approach offered in [28] is likely to reduce the spread of errors for a given ESN configuration. Another approach to consider is the reservoir regularisation approach defined in [62]. Here, it is shown that a combination of Tikhonov, and reservoir regularisation can be used to improve the stability of ESN's with feedback.

As such, changing the range of the output feedback weights should be considered in future work. Also, an implementation of the design approach offered in [28] may lower the errors observed in this work.

5.1.13 The Effects of Precision

The Echo State Network implementation presented here is capable of both double and single precision. While the results presented in Section 4.2 compare the speed performance of the two implementations, the effects of precision on accuracy and learning were not investigated. This would necessitate a review of the existing literature on this topic, and may provide a basis for original work in the specific case of the Echo State Network.

5.1.14 Alternative Building & Training Methods

This GPU Echo State Network implements the ESN building method as proposed in [13], and the offline Tikhonov regularisation training method described in [15, 2]. As seen from the prediction results in Section 4.3.4, the median ESN error can

have a large range when output feedback is used. This influences the number of trials that an ESN designer must perform before a "best" network is found. Further to this, the designer can only use this implementation for offline training problems. More specifically, for problems that are independent and identically distributed, and with sufficient historical information to describe the input-output relationship (see Section 2.1.1). The literature describes several approaches that may offer improvements.

When building an ESN, the implementation could take into account the "richness" measure described in Section 2.2.3.1. This systems-theory approach to ESN design proposes that an ESN has the "richest" set of dynamics when the poles of this ESN are evenly distributed throughout the unit circle. While this would not entirely eliminate the "trial and error" approach to ESN building described above, it would likely reduce the number of trials required to find an ESN well suited to the given problem.

As well as an alternative to building Echo State Networks, the literature presents online alternatives to the Tikhonov regularisation method implemented here. These methods include recursive least squares, and backpropagation-decorrelation (see Section 2.2.5). The latter case is more correctly considered as an alternative reservoir computing approach, rather than an ESN training method. However, both methods offer an online reservoir computer training approach that would benefit the users of this GPU implementation.

5.1.15 General Case Performance Comparison

The performance figures presented in Section 4.2.4 give speed results for an Intel i7-980 CPU and an Nvidia GTX480 (see Section 4.1). While this information is useful for this hardware configuration, it is not sufficient to make a general case statement.

For the ESN designer with a higher specification CPU, and a lower specification GPU, it is not clear whether this GPU implementation will offer them an advantage over a CPU implementation for the problem they may have in mind.

A general statement about GPU and CPU implementations may be obtained empirically. An empirical analysis would require that these tests are run on multiple CPU and GPU platforms. A potentially costly exercise for any single researcher, but one they may be achieved by releasing the libraries with the speed tests, and by asking users to run these tests and submit their results. Those users who run the tests should obtain the results they require for their purposes. If these users in-turn report their results, those users with similar hardware may use the reported results to make a decision.

5.1.16 Cross-Platform Implementation

This implementation was designed for and built using the Nvidia Cuda toolchain. Alternative GPU computing environments are available. Extending this implementation to use, for example, OpenCL (see Section 2.3) would allow users of AMD and other hardware to also use this tool. [34]

5.1.17 A Higher-Level Interface

The libraries that implement this GPU Echo State Network are designed for use by C^{++} programmers. While the users interested in a GPU ESN implementation are likely to have programming skills, providing an alternative interface to these libraries should allow a wider range of users to use them. Potential solutions to this problem would be, for example, Python or Octave programming wrappers. Alternatively, a desktop graphical user interface could be provided. For those wishing to centralise

the tool for use on a GPU server, a web-interface may also be suitable. Obtaining feedback from ESN users and researchers would be useful in guiding such an effort.

5.1.18 Multiple GPU Devices

The implementation presented here was designed to operate on a single GPU. Given a very large Echo State Network, it may be possible to reduce the execution time and share the problem across multiple GPU devices. This would necessitate research into the viability of large (say 1,000,000 reservoir neurons) ESNs as problem solvers, and methods for dividing matrix operations across multiple GPUs. This work may also provide cost savings to users of smaller ESNs that would prefer to purchase multiple lower-end GPUs over one top-end GPU.

References

[1] P. F. Dominey, "Complex sensory-motor sequence learning based on recurrent state representation and reinforcement learning," *Biological Cybernetics*, vol. 73, pp. 265–274, 1995.

[2] M. Lukoševičius and H. Jaeger, "Reservoir computing approaches to recurrent neural network training," *Computer Science Review*, vol. 3, no. 3, pp. 127–149, Aug. 2009.

[3] H. Jaeger, "The 'echo state' approach to analysing and training recurrent neural networks," GMD - German National Research Institute for Computer Science, GMD Report 148, December 2001.

[4] W. Maass, T. Natschlager, and H. Markram, "Real-time computing without stable states: a new framework for neural computation based on perturbations," *Neural Computation*, vol. 14, no. 11, pp. 2531–60, 2002.

[5] H. Jaeger and H. Haas, "Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication," *Science*, vol. 304, pp. 78–80, 2004.

[6] F. Schürmann, K. Meier, and J. Schemmel, "Edge of chaos computation in mixed-mode vlsi - "a hard liquid"," in *NIPS*, 2004.

[7] A. Atiya and A. Parlos, "New results on recurrent network training: unifying

the algorithms and accelerating convergence," *Neural Networks, IEEE Transactions on*, vol. 11, no. 3, pp. 697 –709, May 2000.

[8] J. Steil, "Backpropagation-decorrelation: online recurrent learning with O(N) complexity," in *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, vol. 2, 2004, pp. 843 – 848 vol.2.

[9] W. S. Mcculloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysic*, vol. 5, pp. 115–133, 1943.

[10] S. J. Weddell and R. Y. Webb, "Reservoir computing for prediction of the spatially-variant point spread function," *Selected Topics in Signal Processing, IEEE Journal of*, vol. 2, no. 5, pp. 624–634, Oct. 2008.

[11] P. J. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, Harvard University, 1974.

[12] P. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550 –1560, oct 1990.

[13] H. Jaeger, "Tutorial on training recurrent neural networks, covering BPTT, RTRL, EKF and the "echo state network" approach," German National Research Center for Information Technology, Technical Report 159, October 2002.

[14] A. N. Tikhonov, "Solution of incorrectly formulated problems and the regularization method," *Soviet Math. Dokl.*, vol. 4, pp. 1035–1038, 1963.

[15] F. Wyffels, B. Schrauwen, and D. Stroobandt, "Stable output feedback in reservoir computing using ridge regression," in *International Conference on Artificial Neural Networks*, 2008.

[16] F. Schürmann, K. Meier, and J. Schemmel, "Edge of chaos computation in mixed-mode vlsi - "a hard liquid"," in *In Proc. of NIPS*. MIT Press, 2005.

[17] B. Schrauwen, M. D'Haene, D. Verstraeten, and J. Van Campenhout, "Compact hardware for real-time speech recognition using a liquid state machine," in *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*, 12-17 2007, pp. 1097 –1102.

[18] B. Schrauwen, M. D'Haene, D. Verstraeten, and D. Stroobandt, "Compact hardware liquid state machines on fpga for real-time speech recognition," *Neural Networks*, no. 21, pp. 511–523, 1 2008.

[19] K.-S. Oh and K. Jung, "Gpu implementation of neural networks," *Pattern Recognition*, vol. 37, no. 6, pp. 1311–1314, 2004.

[20] E. Xie, M. McGinnity, Q. Wu, J. Cai, and R. Cai, "Gpu implementation of spiking neural networks for color image segmentation," in *Image and Signal Processing (CISP), 2011 4th International Congress on*, vol. 3, oct. 2011, pp. 1246 –1250.

[21] V. Pallipuram, M. Bhuiyan, and M. Smith, "Evaluation of gpu architectures using spiking neural networks," in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, july 2011, pp. 93 –102.

[22] T. Keith, S. Weddell, and T. Van Cutsem, "Gpu implementation of an echo state network for optical wavefront prediction," in *Proceedings of the Work in Progress Session, 20th Euromicro Intl. Conf. on Parallel, Distributed & Network-based Processing, Garching, Germany*, E. Grosspietsch and K. Klöckner, Eds. SEA-Publications, Johannes Kepler University, Austria, February 2012.

[23] NVIDIA Corporation, *CUDA Toolkit 4.2 CUBLAS Library.* Santa Clara, CA, USA: NVIDIA Corporation, February 2012, version 4.2.

[24] ——, *CUDA Toolkit 4.2 CUSPARSE Library.* Santa Clara, CA, USA: NVIDIA Corporation, February 2012, version 4.2.

[25] ——, *CUDA Toolkit 4.2 CURAND Library.* Santa Clara, CA, USA: NVIDIA Corporation, March 2012, version 4.2.

[26] M. T. Hagan, H. B. Demuth, and M. Beale, *Neural network design.* Boston, MA, USA: PWS Publishing Co., 1996.

[27] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach (3rd Edition)*, 3rd ed. Prentice Hall, December 2010.

[28] M. C. Ozturk, D. Xu, and J. C. Príncipe, "Analysis and design of echo state networks," *Neural Comput.*, vol. 19, no. 1, pp. 111–138, Jan. 2007.

[29] M. Buehner and P. Young, "A tighter bound for the echo state property," *IEEE Transactions on Neural Networks*, vol. 17, no. 3, pp. 820 – 824, May 2006.

[30] J. Schmidhuber, D. Wierstra, M. Gagliolo, and F. Gomez, "Training recurrent networks by evolino," *NEURAL COMPUTATION*, vol. 19, p. 2007, 2007.

[31] D. P. O'Leary, "Near-optimal parameters for tikhonov and other regularization methods," *SIAM J. Sci. Comput.*, vol. 23, no. 4, pp. 1161–1171, Apr. 2001.

[32] J. E. Moody, "The effective number of parameters: An analysis of generalization and regularization in nonlinear learning systems," in *NIPS*, 1991, pp. 847–854.

[33] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, pp. 532–533, 1988.

[34] Khronos Group. (2012, October) OpenCL - The open standard for parallel programming of heterogeneous systems. [Online]. Available: http://www.khronos.org/opencl/

[35] Advanced Micro Devices. (2012, October) Heterogenous computing. Advanced

Micro Devices. [Online]. Available: http://developer.amd.com/TOOLS/HC/Pages/default.aspx

[36] Nvidia Corporation. (2012, October) Nvidia cuda. Nvidia Corporation. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[37] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*. Santa Clara, CA, USA: NVIDIA Corporation, April 2012, version 4.2.

[38] M. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Comput.*, vol. C-21, pp. 948+, 1972.

[39] P. R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*. Springer US, 2010.

[40] (2012, August) Blas. The Netlib Repository at Univ. of Tennesse Knoxville & Oak Ridge National Lab. [Online]. Available: http://www.netlib.org/blas/

[41] (2012, August) Lapack. The Netlib Repository at Univ. of Tennesse Knoxville & Oak Ridge National Lab. [Online]. Available: http://www.netlib.org/lapack/

[42] (2012, June) Magma-1.2.1. The Innovative Computing Lab., Univ. of Tennesse Knoxville. [Online]. Available: http://icl.cs.utk.edu/projectsfiles/magma/magma-1.2.1.tar.gz

[43] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3rd ed. New York, NY, USA: Cambridge University Press, 2007.

[44] G. Golub and W. Kahan, "Calculating the singular values and pseudo-inverse of a matrix," *Journal of the Society for Industrial and Applied Mathematics: Series B, Numerical Analysis*, vol. 2, no. 2, pp. pp. 205–224, 1965.

[45] C. Vömel, S. Tomov, and J. Dongarra, "Divide and conquer on hybrid GPU-

accelerated multicore systems," *SIAM Journal on Scientific Computing*, vol. 34(2), pp. C70–C82, April 2012.

[46] P. R. Willems, B. Lang, and C. Vömel, "Lapack working note 166: Computing the bidiagonal svd using multiple relatively robust representations," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-05-1376, 2005.

[47] C. Lessig and P. Bientinesi, "On parallelizing the mrrr algorithm for data-parallel coprocessors," in *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, ser. PPAM'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 396–402.

[48] S. Lahabar and P. J. Narayanan, "Singular value decomposition on GPU using CUDA," in *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10.

[49] S. Lahabar, "Exploiting the graphics hardware to solve two compute intensive problems: Singular value decomposition and ray tracing parametric patches," Master's thesis, Center for Visual Information Technology International Institute of Information Technology, Hyderabad - 500032, India, August 2010.

[50] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[51] C. M. Bishop, *Neural Networks for Pattern Recognition*. New York, NY, USA: Oxford University Press, Inc., 1995.

[52] M. Lukoševičius, "Echo state networks with trained feedbacks," Jacobs University Bremen, Technical Report 4, February 2007.

[53] R. J. Hyndman and Y. Fan, "Sample quantiles in statistical packages," *The American Statistician*, vol. 50, pp. 361–365, 1996.

[54] J. W. Eaton, D. Bateman, and S. Hauberg, *GNU Octave: A high-level interactive language for numerical computations*, 3rd ed. Boston, MA, USA: Free Software Foundation, Inc., February 2011, version 3.6.1.

[55] J. J. Steil, "Several ways to solve the mso problem." in *ESANN*, 2007, pp. 489–494.

[56] T. Keith and S. J. Weddell, "The Echo State Network on the Graphics Processing Unit," in *Proceedings of the 12th International Conference on Aritificial Intelligence & Soft Computing*. Springer-Verlag, (in press) 2013.

[57] (2012, August) Automatically tuned linear algebra software (atlas). [Online]. Available: http://math-atlas.sourceforge.net/

[58] M. C. Mackey and L. Glass, "Oscillation and chaos in physiological control systems," *Science*, vol. 197, no. 4300, pp. pp. 287–289, 1977.

[59] K. Atkinson, W. Han, and D. Stewart, *Numerical Solution of Ordinary Differential Equations*, ser. Pure and Applied Math. Wiley, 2011.

[60] D. Wierstra, F. J. Gomez, and J. Schmidhuber, "Modeling systems with internal state using Evolino," in *Proc. of the 2005 conference on genetic and evolutionary computation (GECCO), Washington, D. C.* ACM Press, New York, NY, USA, 2005, pp. 1795–1802.

[61] (2012, August) Intel math kernel library. Intel Corporation. [Online]. Available: http://software.intel.com/en-us/intel-mkl

[62] R. F. Reinhart and J. J. Steil, "Reservoir regularization stabilizes learning of echo state networks with output feedback," in *In Proc. ESANN*, 2011.