The Echo State Network on the Graphics Processing Unit

Tūreiti Keith and Stephen J. Weddell

Dept. of Electrical & Computer Engineering, University of Canterbury, New Zealand
tureiti.keith@gmail.com, steve.weddell@canterbury.ac.nz

Abstract. Extending on previous work, the Echo State Network (ESN) and Tikhonov Regularisation (TR) training algorithms were implemented for both the CPU, an Intel i7-980; and the GPU, an Nvidia GTX480. The implementation used all 4 cores of the CPU, and all 480 cores of the GPU. The execution times of these implementations were measured and compared. In the ESN case, speed-ups were observed at reservoir sizes greater than 1,024. The first significant speed-ups of 6 and and 5 were observed at a reservoir size of 2,048 in double and single precision respectively. In the case of Tikhonov Regularisation, no significant speed-ups were observed.

Keywords: echo state network, GPU, Tikhonov regularisation.

1 Introduction

The Echo State Network (ESN) was introduced by Jaeger in 2001 [1]. In 2002, Maass conceived of the Liquid State Machine (LSM) [2]. These efforts mark the beginning of what is now referred to as Reservoir Computing [3]. At the core of a Reservoir Computer (RC) is a randomly generated dynamical system – a *reservoir* of dynamics. The output of an RC is a linear combination of signals tapped from this reservoir. An RC may also (but not necessarily) accept inputs which perturb the reservoir. [1, 2, 3]

This work extends on previous work to implement the Echo State Network on the Graphics Processing Unit (GPU) [4]. The following sections detail the implementation of the ESN and an offline training algorithm based on Tikhonov Regularisation. Section 2 presents the Echo State Network, the Tikhonov Regularisation algorithm, and the GPU. Details of the GPU implementation are given in Sect. 3, and the method for testing the behaviour of this implementation in Sect. 4. The results of these experiments are given in Sect. 5.

2 Background

2.1 The Echo State Network

The Echo State Network (ESN) is a form of Recurrent Neural Network (RNN) topology that lends itself to offline training via linear regression. At the core of an

L. Rutkowski et al. (Eds.): ICAISC 2013, Part I, LNAI 7894, pp. 96–107, 2013.
© Springer-Verlag Berlin Heidelberg 2013

ESN is a sparse, randomly connected RNN comprising sigmoidal neurons. This is referred to as the "reservoir". The output of the ESN is composed of linear neurons that tap signals from the RNN. It is the linear output that facilitates ESN training via linear regression. [1, 3]

Equations (1) & (2) define the behaviour of the network. Equation (1),

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

describes the state of the neurons in the reservoir $\mathbf{x}$ at time n after receiving an input $\mathbf{u}(n-1)$. Here $\mathbf{W}_{\text{in}}$ represents the input weights, $\mathbf{W}$ the reservoir weights, $\mathbf{W}_{\text{ofb}}$ the optional output feedback, and $f(\ldots)$ a sigmoidal activation function. The state of the reservoir is then used in (2),

$$\mathbf{y}(n) = f_{\text{out}}\left(\mathbf{W}_{\text{out}}\begin{bmatrix}\mathbf{u}(n)\\ \mathbf{x}(n)\end{bmatrix}\right), \tag{2}$$

to calculate the ESN output $\mathbf{y}(n)$. Here $\mathbf{W}_{\text{out}}$ are the output weights and $f_{\text{out}}(\ldots)$ is the linear activation function. [1]

At the core of the Echo state Network is its reservoir, represented by the weight matrix $\mathbf{W}$ given in (1). This is a sparse and randomly generated matrix. [1]. Equation (3),

$$\rho = \max\left(|\lambda(\mathbf{W})|\right), \tag{3}$$

gives the spectral radius of $\mathbf{W}$. This is the largest absolute eigenvalue of $\mathbf{W}$, and is typically scaled to a value less than, but close to one. Equation (4),

$$\mathbf{W} = \frac{\rho\mathbf{W}_{\text{rand}}}{\max\left(|\lambda(\mathbf{W}_{\text{rand}})|\right)}, \tag{4}$$

can be used to scale a random matrix, $\mathbf{W}_{\text{rand}}$, to a desired spectral radius, ρ. [1, 5]

Linear regression can be used to train an Echo State Network offline. Only the output matrix, $\mathbf{W}_{\text{out}}$, is trained. One approach to obtain the output matrix $\mathbf{W}_{\text{out}}$ is a form of linear regression known as Tikhonov Regularisation (TR) or Ridge Regression. Equation (5),

$$\mathbf{W}_{\text{out}} = \mathbf{Y}_{\text{target}}\mathbf{X}^T\left(\mathbf{X}\mathbf{X}^T + \alpha^2\mathbf{I}\right)^{-1}, \tag{5}$$

describes TR used to calculate $\mathbf{W}_{\text{out}}$. Here $\mathbf{Y}_{\text{target}}$ is the training target, α is a regularisation constant and $\mathbf{X}$,

$$\mathbf{X} = \begin{bmatrix}\mathbf{u}(1)\ldots\mathbf{u}(n)\ldots\mathbf{u}(N)\\ \mathbf{x}(1)\ldots\mathbf{x}(n)\ldots\mathbf{x}(N)\end{bmatrix}, \tag{6}$$

is a history of the state vector and input vector for N time steps collected while processing training input data with (1). [1, 3, 5, 6]

Equation (5) requires a square matrix inversion. The matrix may not be invertible, however, singular value decomposition can be used to find a pseudo-inverse. [7, 8]

2.2 The General Purpose Graphics Processing Unit

The General Purpose Graphics Processing Unit (GPGPU, or just GPU) is a highly parallel computing platform available to desktop and laptop users. Toolkits for working with GPUs include the open and cross-platform OpenCL; Nvidia's Cuda [9]; and AMD's Heterogeneous Computing Platform, the GPU component is based on OpenCL. Due to the availability of an Nvidia platform with Cuda-based Blas and Sparse mathematics libraries, Nvidia hardware and tools were used in this work. The Nvidia GPU and the Cuda programming model are briefly described in this section.

The Nvidia Cuda Programming Model. Nvidia refers to their GPGPUs as single instruction multiple thread devices (SIMT). The same instruction is executed in parallel on different pieces of data, as per Flynn's single instruction multiple device (SIMD) architecture. However, the SIMT model also allows for conditional branching[1]. On an Nvidia Cuda device, a programmer writes a *kernel* of code that defines these instructions. [10, 11]

On execution, a kernel is run in multiple SIMT threads. Each thread has access to private local memory, shared memory visible to a group of threads (called a *thread block*), and global memory. At run time, a kernel has access to the thread's *thread index*. This can be used to determine which addresses of memory to access. The thread index is a vector of up to three dimensions, and is unique for each thread within the thread block. The dimensionality of the thread index and, therefore, the thread block, allows the programmer to model vector, 1D; matrix, 2D; or volume, 3D calculations.

3 Implementation Details

3.1 The Toolchain

Two Echo State Network and Tikhonov Regularisation implementations were built, one targeting the CPU, the other the GPU. Table 1 summarises the tools used for each implementation. The remainder of this section details a selection of these tools.

Cuda C^{++}. This is a programming language designed for use with the Nvidia Cuda toolchain. It is an extension of a subset of the existing C^{++} ISO/IEC 14882:2003 standard. The primary goal of this language is to facilitate the programming of Single Instruction Multiple Thread (SIMT) code. Using Cuda C^{++}, a developer can compose *kernels*. These kernels are executed on the GPU with instruction-level parallelism across multiple threads (See Sect. 2.2).

[1] Such branching typically impacts negatively on the efficiency of the GPU. [10]

Table 1. ESN and GPU implementation tools

	GPU		CPU	
	ESN	TR	ESN	TR
Language	C^{++}, Cuda C^{++} r4.2		GNU Octave	
Compiler	gcc 4.6.2, nvcc r4.2		Interpreted	
Libraries	Cuda, Cublas, Cusparse, Curand			
		Magma 1.2		
		Atlas 3.8.4		
		Blas 3.3.1, Lapack 3.3.1		

The Cuda Libraries. These are distributed gratis with the Nvidia drivers. The libraries used for this project include release 4.2 of the Cuda, Cublas, Cusparse, and Curand libraries. The Cuda library coupled with the Nvidia Cuda Compiler facilitate the use of C^{++} language level extensions for kernel development. The Cublas library provides GPU implementations of the well known Blas level 1, 2, and 3 routines. The Cusparse library provides GPU implementations of some sparse matrix storage formats and operations. The Curand library provides pseudo-random number generation routines.

The Nvidia Cuda Compiler. Also known as nvcc, the Nvidia Cuda Compiler is used to compile Cuda C^{++} code. It is capable of producing both architecture specific, and compute-capability dependent code. In the latter case, the compiler generates a first-pass compilation, preparing a distributable for Just-In-Time (JIT) compilation. JIT compilation occurs on first execution of the code on the target GPU. With the correct settings on the target PC, the resulting JIT compiled binaries are cached for later use, and are updated with a change to the Nvidia drivers.

The Atlas Library. This is also known as the Automatically Tuned Linear Algebra Software library. It is a free software project to provide tuned Blas and Lapack routines. In this instance, the Atlas library interfaces with the reference Fortran77 implementation of Blas, and its accompanying Lapack implementation.

The Magma Library. Magma is a free software project to migrate Lapack routines to the GPU. Magma currently implements hybrid CPU/GPU versions of Lapack routines calling, in this instance, a mixture of Cublas, Magma Lapack, and the Fortran77 reference Lapack routines.

GNU Octave. Octave is a high-level interpreted programming language for linear algebra. The GNU Octave environment/interpreter is free software. In this instance Octave interfaces with the Atlas library to perform linear algebraic operations.

3.2 Implementing the ESN and TR Algorithms

The Echo State Network and Tikhonov Regularisation algorithms were implemented for the GPU and CPU using the toolchain described in Sect. 3.1. Blas, Sparse, Lapack, and Pseudo Random Number libraries, along with bespoke kernels were used to implement (1), (2), (3), (4),(5), and (6). The details of the GPU implementation follow, and are summarised in Table 2.

Table 2. ESN and GPU implementation details

	Operation	Implementation
Reservoir generation, (3) & (4)	$\mathbf{W}_{\text{rand}}$	Curand pseudo-random number generator
	$\lambda\left(\mathbf{W}_{\text{rand}}\right)$	Magma eigenvalue extraction
	$\frac{\rho \mathbf{W}_{\text{rand}}}{\dots}$	Cublas scalar-vector multiplication
ESN state calculation, (1)	$\mathbf{W}_{\text{in}}\mathbf{u}\left(n\right)$, $\mathbf{W}_{\text{ofb}}\mathbf{y}\left(n-1\right)$	Cublas matrix-vector multiplication
	$\mathbf{W}\mathbf{x}\left(n-1\right)$	Cusparse matrix-vector multiplication (with $\mathbf{W}$ stored in compressed sparse row, CSR, format [12, 13])
	$f\left(\ldots + \ldots + \ldots\right)$	Single bespoke kernel
ESN output calculation, (2)	$\begin{bmatrix} \mathbf{u}\left(n\right) \\ \mathbf{x}\left(n\right) \end{bmatrix}$	Cuda memory copy
	$f_{\text{out}}\left(\mathbf{W}_{\text{out}}\left[\ldots\right]\right)$	Cublas matrix-vector multiplication (f_{out} is linear)
Tikhonov Regularisation, (5)	$\mathbf{X}\mathbf{X}^T + \alpha^2\mathbf{I}$	Cublas matrix-matrix & scalar-vector multiplication, vector addition
	$\left(\ldots\right)^{-1}$	Magma SVD, Cublas matrix-matrix multiplication, bespoke diagonal matrix inversion kernel
	$\mathbf{Y}_{\text{target}}\mathbf{X}^T\left(\ldots\right)$	Cublas matrix-matrix multiplication

Calculating the ESN Reservoir State. Described in (1), this implementation uses Cublas matrix-vector multiplication to perform $\mathbf{W}_{\text{in}}\mathbf{u}\left(n\right)$ and $\mathbf{W}_{\text{ofb}}\mathbf{y}(n-1)$. As $\mathbf{W}$ is sparse (see Sect. 2.1), $\mathbf{W}\mathbf{x}\left(n-1\right)$ is performed using Cusparse matrix-vector multiplication. $\mathbf{W}$ is stored in compressed sparse row (CSR) format [12, 13]. A single bespoke kernel executes the activation function $f\left(\ldots\right)$, and the addition of factors $\mathbf{W}_{\text{in}}\mathbf{u}\left(n\right)$, $\mathbf{W}\mathbf{x}\left(n-1\right)$, and $\mathbf{W}_{\text{ofb}}\mathbf{y}\left(n-1\right)$. Here, $f\left(\ldots\right)$ is a hyperbolic tangent. When a user initiates an ESN reservoir state calculation, they can choose to provide multiple time-steps of input data, e.g. training data, and collect the input and reservoir state vectors in host memory as per (6). This state history can be later used to train the ESN as per (5).

There is a limit to the amount of input data that may be used, this is dependent on the size of the ESN, and the memory available on the GPU. Similarly,

if the reservoir state vector history is to be used for training, then the size of the history will be limited by the size of the GPU memory, and the accompanying matrices described in (5). The implementation does not currently warn of memory limitations.

Calculating the ESN Output. The ESN output, (2), is calculated using Cublas and Cuda memory copy operations. A Cuda device-side memory copy is used to stack the vectors $\mathbf{u}(n)$ and $\mathbf{x}(n)$. A Cublas matrix-vector multiplication is used to perform

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

The Tikhonov Regularisation Algorithm. Described in (5), this was implemented using Cublas and Magma libraries. Blas matrix-matrix and scalar-vector multiplications were used to obtain $\left(\mathbf{X}\mathbf{X}^T + \alpha^2\mathbf{I}\right)$. To invert this result, singular value decomposition is used [8, 7]. This was implemented using Magma provided SVD, Cublas matrix-matrix multiplication, and a bespoke diagonal-matrix pseudo-inverse kernel.

Reservoir Generation. To generate a reservoir of a given spectral radius and connectivity, (3) & (4) were implemented. This required the pseudo-random number generating library, Curand, to create $\mathbf{W}_{\text{rand}}$ with the desired connectivity, and a Magma routine to extract its eigen-values. Cublas scalar-vector multiplication was used to scale $\mathbf{W}_{\text{rand}}$ as per (4).

4 Experimental Configuration

The goal of these experiments is to gather information that will help users decide when best to perform Echo State Network and Tikhonov Regularisation algorithms on a GPU, and when best to use a CPU. Four experiments have been devised, two of which have been executed. Two further experiments are described in Sect. 7. The two executed experiments examine the relative speed performance of CPU and GPU based ESN and TR algorithms in both double and single precision. The methods and equipment used to perform this evaluation follow.

4.1 Hardware

To perform this comparison, a multi-core Intel Core i7-920 was used as the CPU, and an Nvidia GTX480 as the GPU. Both are representative of high-end commodity hardware in their respective domains. Comparative information on these processors is presented in Table 3.

Table 3. 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	–	1
Memory[2]	6 GiB	1.5 GiB
Memory clock speed	1.066 GHz	1.848 GHz
Shared memory per block	–	48 KiB
PCI bus speed[3]	–	2.5 GiT/s

4.2 Measurement Variables

Selected variables were isolated to measure the Echo State Network and Tikhonov Regularisation speed performance. To measure ESN speed performance, three independent variables were isolated – reservoir size, calculation precision and hardware type. The same independent variables were isolated for TR speed performance measurements, with an additional fourth variable, the number of execution time-steps. This is the number of columns, N, in (6). The remaining variables were controlled. A summary of the values used in the experiment is given in Table 4, where irrelevant variables are indicated with a "–".

Table 4. ESN and TR speed comparison variables

Variable	ESN Values	TR Values
Hardware	{Intel i7-980, Nvidia GTX-480}	
Calculation precision	{double, single}	
ESN reservoir size	$\{2^4, 2^5, \ldots, 2^{11}\}$	
ESN execution time steps	2^{10}	$\{2^4, 2^5, \ldots, 2^{16}\}$
ESN input size	2^4	
ESN output size	2^4	
ESN output feedback	present	–
ESN reservoir connectivity	10 %	–
ESN reservoir spectral radius (ρ)	0.9	–
Tikhonov regularisation factor (α)	–	0.1

4.3 Measurement Method

To ensure statistically valid measurements, each timing measurement was repeated 20 times. The first 10 timing measurements were discarded to reduce

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

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

the impact of any just-in-time compiled elements. For each point in independent variable space, a mean and standard deviation execution time was recorded.

5 Results

The execution times of the Echo State Network are given in Fig. 1. Tikhonov Regularisation executions times are given in Figs. 2 & 3. From these measurements, ESN and TR mean speed-up times are presented in Tables 5 & 6 respectively.

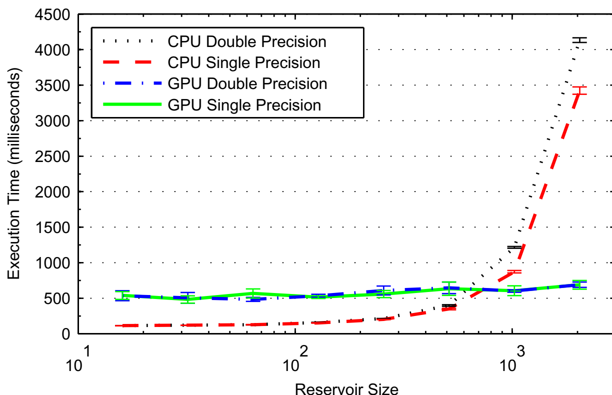

Fig. 1. Mean Echo State Network execution times – CPU versus GPU

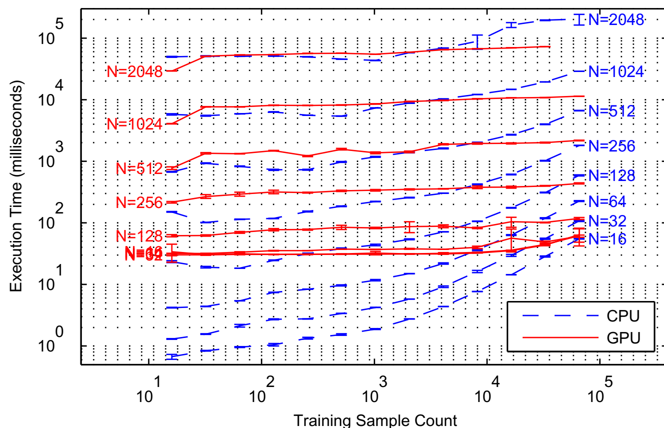

Fig. 2. Double precision Tikhonov Regularisation execution times – CPU versus GPU for varying reservoir sizes (r)

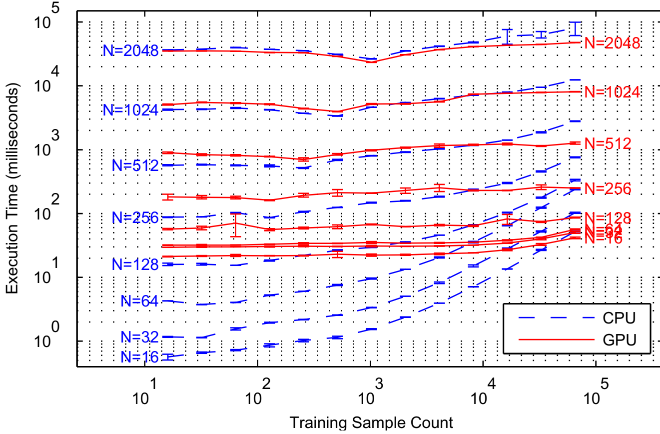

Fig. 3. Single precision Tikhonov Regularisation execution times – CPU versus GPU for varying reservoir sizes (r)

5.1 Echo State Network Speed Performance

In the ESN case, the GPU implementation gives a speed-up at reservoir sizes of 1024 and 2048 (Table 5). The largest speed-up, 5.9923, is observed for a reservoir size of 2048 in double precision. The largest slow-down is 0.2107 at a reservoir size of 16 in single precision.

Table 5. Echo State Network execution: Observed GPU speed-up. The largest and smallest speed-ups are given in bold.

Reservoir Size	ESN Execution: ESN Speed-up Double Precision	Single Precision
16	$\mathbf{0.2130 \pm 0.1314}$	$\mathbf{0.2107 \pm 0.1048}$
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	$\mathbf{5.9923 \pm 0.0563}$	$\mathbf{4.9652 \pm 0.0893}$

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 ESNs 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. GPU thread occupancy and the impact of memory transfers is yet to be measured.

5.2 Tikhonov Regularisation Speed Performance

In the TR case, the speed-up of the extreme reservoir sizes was calculated. The speed-ups for $r = 16$ and $r = 2048$ are given in Table 6.

Table 6. Tikhonov Regularisation execution: Observed GPU speed-up. The largest and smallest speed-ups are given in bold.

History Size	TR Execution: GPU Speed-up			
	$r = 2048$		$r = 16$	
	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	–	$\mathbf{1.6864 \pm 0.2362}$	$\mathbf{0.8561 \pm 0.2424}$	$\mathbf{1.2571 \pm 0.0289}$

In the $r = 16$ case one speed-up of 1.2571 occurred at a history size of 65,536 in single precision, all other measures gave a slow-down. The largest slow-down, 0.0197, was observed for a history size of 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, implying that the variability of measurements at these points is too high to give an accurate measure.

In the $r = 2048$ case speed-ups were observed at most measurement points, excluding from a history size of 64 to 2,048 in the double precision case. The greatest speed-up, 2.6813, was observed at a history size of 32,768 in double precision. The largest single precision speed-up, 1.6864 was observed at a history size of 65,536. The greatest slow-down 0.7910 was observed at a history size of 1024 in the double precision case. It should be noted that in the $r = 2048$, double precision case, the measurement at history size 65,536 could not be taken as the GPU had reached its global memory limits.

The slow-down observed may be partly attributed to host-GPU memory transfers that take place. The current implementation uses Magma's SVD implementation. 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.

6 Conclusion

The Echo State Network and Tikhonov Regularisation training algorithms were implemented for both the CPU, an Intel i7-980; and the GPU, an Nvidia GTX480. The execution times of these implementations were measured and compared.

In the ESN case, speed-ups were observed at reservoir sizes greater than 1,024. The first significant speed-ups of 5.9923 and 4.9652 were observed at a reservoir size of 2,048 in double and single precision respectively.

In the case of Tikhonov Regularisation, no significant speed-ups were observed, and memory limitations were seen for large reservoir state history sizes. This may be due to host-GPU memory transfers required to perform singular value decomposition.

7 Future Work

Experimental refinement, two further experiments, and profiling work is planned. Large variations are observed at some measurement points (See Sect. 5.2), which warrants further investigation. This may be an indication that the measurement "warm-up" time was insufficient. Two further experiments will be conducted. These aim to compare the execution times of the CPU and GPU implementations as reservoir connectivity changes, and when performing a full train-test cycle on chaotic time-series data. Finally, the GPU implementation will be profiled. This may yield information on inefficiencies in the design of the program, and therefore guide us to points of optimisation.

References

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

[2] Maass, W., Natschlager, T., Markram, H.: Real-time computing without stable states: a new framework for neural computation based on perturbations. Neural Comput 14(11), 2531–2560 (2002)

[3] Lukoševičius, M., Jaeger, H.: Reservoir computing approaches to recurrent neural network training. Computer Science Review 3(3), 127–149 (2009)

[4] Keith, T., Weddell, S., Van Cutsem, T.: Gpu implementation of an echo state network for optical wavefront prediction. In: Grosspietsch, E., Klöckner, K. (eds.) Proceedings of the Work in Progress Session, 20th Euromicro Intl. Conf. on Parallel, Distributed & Network-based Processing, Garching, Germany. SEA-Publications, Johannes Kepler University, Austria (2012)
[5] Jaeger, H.: 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)
[6] Tikhonov, A.N.: Solution of incorrectly formulated problems and the regularization method. Soviet Math. Dokl. 4, 1035–1038 (1963)
[7] Golub, G., Kahan, W.: Calculating the singular values and pseudo-inverse of a matrix. Journal of the Society for Industrial and Applied Mathematics: Series B, Numerical Analysis 2(2), 205–224 (1965)
[8] Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes 3rd Edition: The Art of Scientific Computing, 3rd edn. Cambridge University Press, New York (2007)
[9] Nvidia Corporation. Nvidia cuda. Nvidia Corporation (October 2012), http://www.nvidia.com/object/cuda_home_new.html
[10] NVIDIA Corporation, NVIDIA CUDA C Programming Guide, version 4.2. NVIDIA Corporation, Santa Clara (April 2012)
[11] Flynn, M.: Some Computer Organizations and Their Effectiveness. IEEE Trans. Comput. C-21, 948+ (1972)
[12] NVIDIA Corporation, CUDA Toolkit 4.2 CURAND Library, version 4.2. NVIDIA Corporation, Santa Clara (March 2012)
[13] Eaton, J.W., Bateman, D., Hauberg, S.: GNU Octave: A high-level interactive language for numerical computations, version 3.6.1, 3rd edn. Free Software Foundation, Inc., Boston (2011)