CUDA Implementation of LSTM for MagmaDNN *

With developer notes for internal review

Pierluigi Cambie-Fabris
University of Tennessee, Knoxville
Knoxville, TN
pcambiefabris@gmail.com

Joshua Zingale
San Diego State University
San Diego, CA
jtzingale@gmail.com

***Abstract*—This paper presents an LSTM implementation on GPU computational platforms for the upcoming version release of MagmaDNN, discussing the mathematical and computational methods employed. These include caching reusable results and support for accelerated, parallelized computation on NVIDIA processing chips though the CUDA API.**

***Index Terms*—lstm, cuda, magma, deep learning**

I. Introduction

A. MagmaDNN Framework

MagmaDNN is an open-source, C++ deep-learning framework, built using MAGMA and CUDA [1], which facilitates time-efficient training and execution through parallelized Graphics Processing Unit (GPU) code. In order to store data in an efficient manner for the given use case, MagmaDNN employs MemoryManager objects to determine how and where to store data. MemoryManager offers 4 options: HOST (main memory), DEVICE (GPU memory), MANAGED (main and GPU memory with MagmaDNN protocol), and CUDA_MANAGED (main and GPU memory with CUDA protocol).

MagmaDNN has four main classes for creating a neural network: Tensor, Operation, Layer, and Model. These classes serve as the different levels of abstraction for constructing a deep-learning model.

B. Tensor

The Tensor class is the primary data-storage unit in MagmaDNN. It utilizes a MemoryManager to store data in either main or GPU memory. A Tensor object offers various initialization methods, a getter and a setter, and a method to access a pointer to its storage address in memory. The data of a Tensor, managed by the MemoryManager, is stored sequentially in memory, allowing CUDA kernels to easily index data stored inside a Tensor.

C. Operation

The Operation class is the superclass from which all Operation classes inherit. Encapsulated within an Operation are its output Tensor, computations to produce it, computations for gradients with respect to its inputs (if any), and linkage code, which identifies any Operation object inputs used in its own computations. Variable is an Operation which wraps directly around a Tensor to store data. Operation classes in MagmaDNN take any number of inputs and use those inputs to produce an output Tensor. In practice, Operation objects are superimposed as arguments into each other (e.g. matmul(add(VarA,VarB),negative(VarC)), constructing a large compute tree from which gradients with respect to its Variable objects can be computed.

D. Layer

The Layer class is the superclass from which all Layer classes inherit. Layer objects take an Operation for input and then have a method to return an Operation, which serves as the output for the Layer. A Layer also has a method that returns a list of trainable weights and biases. Layer objects can be connected by passing one Layer's output to another Layer's input. In practice, a Layer's initialization method receives some input Operation object or objects and then superimposes other operations on top of the input or inputs: this becomes the Layer's output Operation.

E. Model

The Model class is the superclass from which currently only one class inherits, NeuralNetwork. A NeuralNetwork object allows multiple Layer objects to be placed in a sequential order, one Layer's output feeding into the next Layer's input until the last Layer produces the NeuralNetwork's output. A NeuralNetwork accepts arguments to define how its loss is calculated, how it updates the parameters of its Layer objects to minimize the loss, and the number of epochs for training. A NeuralNetwork can then be used, with either trained or set weights, to evaluate inputs, feeding them through all its Layer objects, to produce outputs.

II. CUDA

CUDA is a platform created by NVIDIA that provides parallel-computing capabilities. CUDA allows programs to use the GPU of a computer in order to perform various calculations in parallel, resulting in less of a time requirement for computations [2]. For this reason, CUDA is used in many computationally-intensive applications such as climate modeling and machine learning.

This project was funded by the National Science Foundation.

CUDA is supported for use in a number of programming languages, including C++ and Fortran, in which functions can call CUDA kernels that divide work amongst the many threads and blocks in the GPU, parallelizing computations. Neural Networks, including those with LSTM components, require many independent and repeated calculations in order to train and thus take strong advantage of the parallel computing capabilities of CUDA.

III. MAGMA

MAGMA, Matrix Algebra on GPU and Multicore Architectures, is a lightweight linear algebra package with a focus on using GPU and multicore devices [3]. Using these hardware accelerators, it provides highly optimized linear algebra operations such as matrix multiplication and transposition. MAGMA is built on Fortran and C++. Its functionality is similar to that of LAPACK (Linear Algebra Package) but with GPU capabilities for parallel computing.

IV. LSTM

In the section which follows lie the background information, the mathematical description, and the specific technical and computational methods employed for this LSTM's implementation in MagmaDNN. The mathematical description relates directly to the programmed implementation.

A. Background

Jürgen Schmidhuber and Sepp Hochreiter first proposed the LSTM architecture in 1997 [4]. The LSTM formulation addressed a problem with conventional recurrent neural networks: calculating gradients over sequences would result in either exploding or vanishing gradients. Such an issue made training models to learn long sequence relationships inefficient if not impossible. The original LSTM contended with this problem by introducing a memory cell, which could carry data from earlier sequence steps to later sequence steps. A later by paper by Cummins the again Schmidhuber proposed modifications which further improved LSTM's ability to learn relationships between both far and nearly separated sequence steps, introducing a forget gate, which allows LSTM to learn not only what data to propagate further into networks but to learn also what data to cease propagating [5]. The present paper describes the implementation of this later and improved instantiation of LSTM into MagmaDNN.

B. Preliminary Implementation Details

Creating LSTM in MagmaDNN requires only that two major components be implemented: the forward calculation and the backward (i.e. gradient) calculation. These two components are coded into an Operation. The Layer class then encapsulates this Operation along with some basic logic to determine what parameters, which are Layer weights, need to be updated by the Optimizer, which is a component of the Model in which the LSTM resides. Existent structures in MagmaDNN such as the NeuralNetwork Model and the Adam Optimizer facilitate updating weight values and handling batches of data, leaving only the particulars of LSTM calculations to implement. Thus, the Model utilizing LSTM, for purposes of describing implementation, can be treated as a black box that passes three-dimensional input into the Layer, prompting a return of a two- or three-dimensional output; then, the network passes the two- or three-dimensional gradient of the network loss function with respect to the Layer's output into the Layer's gradient function, prompting a return of the gradient of the loss function with respect to the Layer input and the weights.

Data for inputs, outputs, and weights, which may be represented as variables referencing matrices in the mathematical formulation, are actually stored within n-dimensional Tensor objects. These will be described more in the Implementation section.

C. Mathematical Formulation

For this implementation of LSTM, input vectors are stored as rows in matrices and weight vectors are stored as columns in matrices. Let N, T, K, $M \in \mathbb{N}$ be the batch size, number of sequence steps, number of values per input vector, and number of nodes in an LSTM respectively. For input, the LSTM receives tensor $X \in \mathbb{R}^{N\times T\times K}$. For output, if the LSTM is returning sequences, the LSTM generates tensor $Y \in \mathbb{R}^{N\times T\times M}$; if it is not returning sequences, it generates tensor $Y \in \mathbb{R}^{N\times M}$.

1) Forward: Let $x_t \in \mathbb{R}^{N\times K}$ and $y_t \in \mathbb{R}^{N\times M}$, the latter defined only when returning sequences, be the inputs and outputs for sequence step t, which are defined thus:

$$\begin{aligned}(x_t)_{ij} &= X_{itj},\\ (y_t)_{ij} &= Y_{itj}.\end{aligned}$$

Let c_t, $h_t \in \mathbb{R}^{N\times M}$ be the internal states for the t^{th} sequence step; c_0 and h_0 are both initialized before computation, typically to all zeros. Let W_f,W_i, W_o, $W_c \in \mathbb{R}^{K\times M}$ and U_f, U_i, U_o, $U_c \in \mathbb{R}^{M\times M}$ be weight matrices. Let b_f, b_i, b_o, $b_c \in \mathbb{R}^{N\times M}$ be bias matrices, which are hence considered "weights." Aside from matrix multiplication, which is signified by juxtaposing two matrices, all operations are element-wise. "$\sigma(x)$" is a sigmoid function and "$\otimes$" is multiplication.

The equations for forward propagation are as follows:

$$\begin{aligned}f_t &= \sigma(x_t W_f + h_{t-1} U_f + b_f),\\ i_t &= \sigma(x_t W_i + h_{t-1} U_i + b_i),\\ o_t &= \sigma(x_t W_o + h_{t-1} U_o + b_o),\\ \tilde{c}_t &= \tanh(x_t W_c + h_{t-1} U_c + b_c),\\ c_t &= f_t \otimes c_{t-1} + i_t \otimes \tilde{c}_t,\\ h_t &= o_t \otimes \tanh(c_t).\end{aligned}$$

If the LSTM is not returning sequences, its output Y is h_T; if the LSTM is returning sequences, the output is tensor $H \in \mathbb{R}^{N\times T\times M}$, with

$$H_{itj} = (h_t)_{ij} \text{ for } t \in [1, T].$$

2) Backward: The parameters of the LSTM are W_f, U_f, b_f, W_i, U_i, b_i, W_o, U_o, b_o, W_c, U_c, and b_c. To train these weights, the gradients of the loss with respect to them must be calculated. Additionally, if there are layers previous to the LSTM layer in a network, the gradient must be calculated with respect to the inputs.

Let L be the loss function for this network. Henceforth, for any matrix A, let $\bar{A}$ be a matrix or tensor with equivalent dimensions to A, with

$$\bar{A}_{i_0 i_1 \ldots i_n} = \frac{\delta L}{\delta A_{i_0 i_1 \ldots i_n}}.$$

The calculations below assume that $\bar{Y}$ is provided by the neural network in which the LSTM resides. Now, define the following five variables:

$$\begin{aligned}
\bar{\gamma}_{f_t} &= \bar{c}_t \otimes c_{t-1} \otimes f_t \otimes (1 - f_t), \\
\bar{\gamma}_{i_t} &= \bar{c}_t \otimes \tilde{c}_t \otimes i_t \otimes (1 - i_t), \\
\bar{\gamma}_{o_t} &= \bar{h}_t \otimes \beta_t \otimes o_t \otimes (1 - o_t), \\
\bar{\gamma}_{c_t} &= \bar{c}_t \otimes i_t \otimes (1 - \tilde{c}_t^2), \\
\bar{\eta}_t &= \bar{\gamma}_{f_{t+1}} U_f^\intercal + \bar{\gamma}_{i_{t+1}} U_i^\intercal + \bar{\gamma}_{o_{t+1}} U_o^\intercal + \bar{\gamma}_{c_{t+1}} U_c^\intercal \text{ if } t < T,
\end{aligned}$$

which can be calculated for $t \in [1, T]$ alongside these equations:

$$\begin{aligned}
\beta_t &= \tanh(c_t) \\
\bar{c}_t &= \begin{cases} \bar{h}_t \otimes o_t \otimes (1 - \beta_t^2) & \text{if } t = T, \\ \bar{h}_t \otimes o_t \otimes (1 - \beta_t^2) + f_{t+1} \otimes c_{t+1} & \text{if } t < T; \end{cases} \\
\bar{h}_t &= \begin{cases} \bar{y}_t & \text{if } t = T, \\ \bar{\eta}_t + \bar{y}_t & \text{if } t < T \text{ \& returning sequences}, \\ \bar{\eta}_t & \text{if } t < T \text{ \& not returning sequences}. \end{cases}
\end{aligned}$$

Finally, the gradients with respect to the weights and with respect to the inputs can be calculated in this way, noting that "$\intercal$" is a matrix transpose while T is still the number of sequence steps:

$$\begin{array}{l|l|l}
\bar{W}_f = \sum_{t=1}^T x_t^\intercal \bar{\gamma}_{f_t} & \bar{U}_f = \sum_{t=1}^T h_{t-1}^\intercal \bar{\gamma}_{f_t} & \bar{b}_f = \sum_{t=1}^T \bar{\gamma}_{f_t} \\
\bar{W}_i = \sum_{t=1}^T x_t^\intercal \bar{\gamma}_{i_t} & \bar{U}_i = \sum_{t=1}^T h_{t-1}^\intercal \bar{\gamma}_{i_t} & \bar{b}_i = \sum_{t=1}^T \bar{\gamma}_{i_t} \\
\bar{W}_o = \sum_{t=1}^T x_t^\intercal \bar{\gamma}_{o_t} & \bar{U}_o = \sum_{t=1}^T h_{t-1}^\intercal \bar{\gamma}_{o_t} & \bar{b}_o = \sum_{t=1}^T \bar{\gamma}_{o_t} \\
\bar{W}_c = \sum_{t=1}^T x_t^\intercal \bar{\gamma}_{c_t} & \bar{U}_c = \sum_{t=1}^T h_{t-1}^\intercal \bar{\gamma}_{c_t} & \bar{b}_c = \sum_{t=1}^T \bar{\gamma}_{c_t}
\end{array}$$

$$\bar{x}_t = \bar{\gamma}_{f_t} W_f^\intercal + \bar{\gamma}_{i_t} W_i^\intercal + \bar{\gamma}_{o_t} W_o^\intercal + \bar{\gamma}_{c_t} W_c^\intercal.$$

D. Implementation

Data is passed to and from the LSTM Layer as Tensor objects, which provide access to memory stored on either DEVICE or HOST. However, the current implementation only supports calculations on DEVICE memory because aspects of the CUDA code cannot directly operate upon HOST memory. All variable names herein reference the variables defined in the mathematical formulation, except the variable names should be interpreted to be the names of Tensor objects, storing data equivalent to the values represented by the mathematical variables.

This LSTM implementation requires both the forward and backward calculations. Many of the values calculated during the forward calculation are useful when calculating the backward pass. Hence, these values are stored in variable caches within the LSTM Layer class during the forward calculation. The backward calculation then utilizes these cached values in its execution. This caching process is visualized in Figure 1.

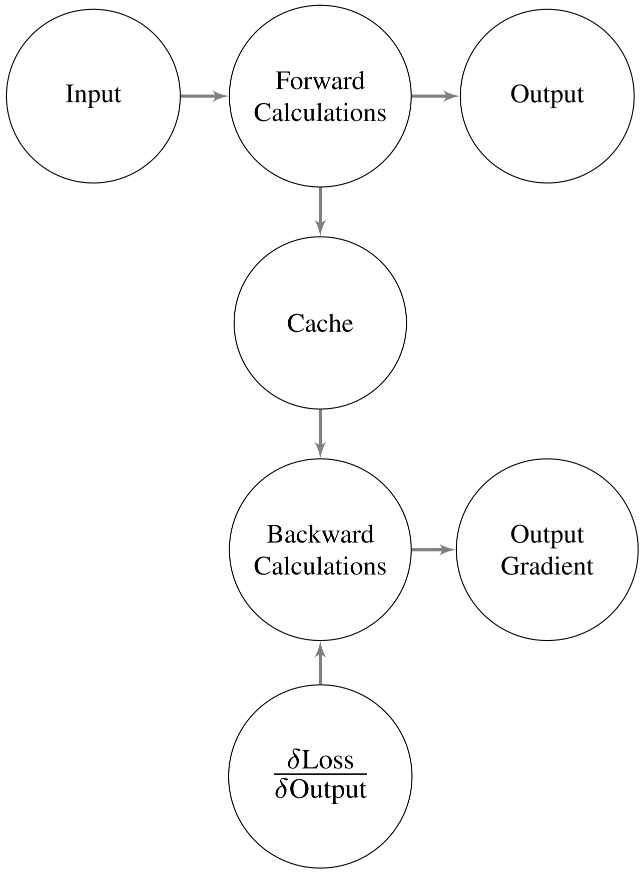

Fig. 1. LSTM Computational Flowchart. The circles represent data sources, destinations, or modifications and edges represent data flow. The graph visualizes how this LSTM implementation reuses cached values from the forward pass in the backward calculation.

Before execution begins, h_0 and c_0 are by default initialized to all zeros. A caller of the Layer may also initialize these Tensor objects to contain particular values. Also, all weight Tensor objects are by default initialized uniformly to contain pseudo-random values between 0 and 1, leaving the option for a caller to set particular initializations.

There are two modes in which an LSTM Layer can be initialized, one for returning sequences, that is all of $h_1 \ldots h_T$, and one for returning only h_T as output, that is, not returning sequences.

1) Forward: The following element-wise operations facilitate the LSTM forward pass calculation on Tensor input: logistic sigmoid, tanh, multiplication, addition, and subtraction. Basic CUDA kernels each run one of these calculations. A MAGMA function calculates the needed matrix multiplications. Finally, there are two special operations implemented as CUDA kernels, slice and concatenate, which assist in accessing data for particular sequence steps and constructing the output sequence respectively.

The LSTM Layer receives a three dimensional Tensor as input, X, from either the input of a Model or from the output of a Layer previous to the LSTM Layer itself. This input is then split into one Tensor per sequence step, which is one of $x_1 \ldots x_T$. To accomplish this, a CUDA kernel copies values at appropriate indices out of Tensor X to populate each Tensor x_t. Next, the various functions previously listed calculate each of $h_1 \ldots h_T$. If the Layer has been initialized to return sequences, the concatenation CUDA kernel concatenates every h_t, returning the output of that concatenation; alternatively, it returns h_T if it is not returning sequences. As stated in Preliminary Implementation Details, many of the intermediate calculations' results are useful for the gradient calculation; so, throughout computation, eight C++ vectors store the values for each sequence step of x_t, c_t, h_t, β_t, f_t, i_t, o_t, and c_t. The forward calculation caches every x_t, so that the backward calculation does not need to re-slice X into its sequence steps.

2) Backward: The backward calculation's efficiency depends upon the eight caches stored in vectors by the forward pass. In the absence of these cached values, the backward calculation would need to recalculate much of what the forward pass already evaluated.

To begin calculation, the Model object which contains this Layer object will call the Layer's gradient function, providing it with $\bar{Y}$ and the variable with respect to which the derivative of the loss function should be calculated. Regardless of the variable with respect to which the gradient is to be returned, a full backward propagation through the sequence is required. This propagation occurs starting at sequence step T and ends once sequence step 1 has been calculated.

A variety of CUDA kernels supports the backward calculations, which differ from those used in the forward calculation. Whereas the forward pass utilizes general kernels that evaluate one operation at a time, those used for the backward pass receive multiple of the cached values alongside some values already computed in the backward pass to implement chains of calculations. There are five specialized kernels to evaluate $\bar{\gamma}_{f_t}$, $\bar{\gamma}_{i_t}$, $\bar{\gamma}_{o_t}$, $\bar{\gamma}_{c_t}$, and $\bar{c}_t$. Slice and concatenate again transform between three-dimensional and two-dimensional data. Additionally, there are two general utility kernels, one for setting all values in a Tensor to 0 and one for summing an arbitrary number of Tensor objects element-wise. As with the forward pass, MAGMA computes the needed matrix multiplications.

If the LSTM Layer is returning sequences, slice works upon $\bar{Y}$ to attain every $\bar{y}_t$. If the LSTM Layer is not returning sequences, $\bar{Y}$, which would already be two-dimensional, need not be sliced and the values thereof initialize $\bar{h}_T$.

Once $\bar{h}_T$ has been calculated, the backward propagation through time begins, with the five specialized kernels running for each sequence step. Depending on the variable with respect to which the gradient is to be calculated, combinations of cached values, the current values of the weights and inputs, and the values computed by this backward pass combine by the appropriate operations to evaluate and then return the gradient.

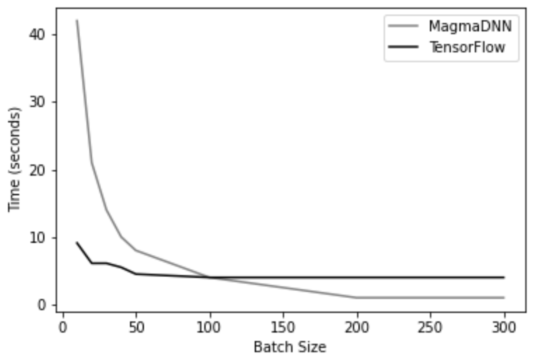

Fig. 2. Training speed of the LSTM in MagmaDNN and the LSTM in TensorFlow compared for various batch sizes.

V. Results

The calculations for this LSTM Layer were compared against a Python script[1] that simulated LSTM computations for both the forward and backward propagation. The LSTM Layer's output values and gradients with respect to the input and each of the parameters matched the expected correct values.

This Layer was also given a basic test case to compare its training speed to that of the LSTM in Google's deep-learning library, TensorFlow [6]. In both MagmaDNN and TensorFlow, we created a model with two layers: a five unit (dimensionality of output space) LSTM that returns sequences connected to a one unit LSTM without returning sequences. The models were given 300 uniform values between 0 and 1 for input and taught to predict only 0's as output in 200 epochs. The training speeds for these models were measured in a Google Colaboratory Notebook running on GPU hardware with varying batch sizes from 10 to 300. The performances can be seen in Figure 2.

The MagmaDNN model performed faster for batch sizes of 100 or above while the TensorFlow model performed faster for the smaller batch sizes.

VI. Conclusion

The LSTM herein presented offers MagmaDNN a needed addition, that it may possess RNN functionality. While there are still changes and improvements to be made before a release version, the initial testing displays promising results toward competitiveness with a leading deep-learning framework.

A. Future Work

More testing on this LSTM is required before firm conclusions about its performance can be drawn. It would be greatly beneficial to test it against established data sets for benchmarking purposes.

In other libraries' implementations of LSTM, such as Tensorflow's [6], sequences of arbitrary length can be used as

[1]The Python script was verified against test cases of known LSTM computation.

input into and returned as output from LSTM. In the current development version, an LSTM's input and output sequence length must be set before training and cannot be changed. So, to achieve these dynamic length capabilities, a user of this LSTM must manually copy the weights of an LSTM trained with one sequence length and then initialize another LSTM with a different sequence length to have these copied weights.

Instead of first using a kernel to slice the sequenced data into two-dimensional Tensor objects and then computing for a sequence step, the CUDA code could be optimized to directly use the three-dimensional input of Tensor objects in its kernels.

In the forward calculation, the computations could be streamlined by combining multiple micro calculation kernels into one. For example, instead of first multiplying with one kernel and then adding with another, the two could be combined to run at a more rapid pace.

In the backward calculation, many of the calculated values needed to determine the derivative with respect to one variable are also used for all others. For instance, every variable requires all of γ_{f_t}, γ_{i_t}, γ_{o_t}, and γ_{c_t} to calculate the derivative with respect to it. MagmaDNN's methods for updating weights is to calculate the derivative with respect to one variable at a time. As it is implemented now, the LSTM does not cache such values and instead recalculates each variable for each derivative. Implementing a cache would greatly reduce recalculating, leading to speed improvements.

VII. Internal Development Notes

A. First Approach

For our first attempt at implementing the LSTM layer, we chose to superimpose a number of previously existing operations with our newly implemented slice and concatenation operations. This approach resulted in correct calculations and successful training on small data sets. However, this implementation had many significant problems. First, our method of superimposing operations, for reasons unknown to us, resulted in the adding of the LSTM to a network taking seemingly exponentially increasing amounts of time for longer (¿10) lengths of sequences. We traced the slow down to some code which calls some custream-related functions in the Operation.h file. These lines of code became slower with each operation that was passed into the next. Our hypothesis is that our network created an amount of operations which was not anticipated when MagmaDNN's Operation class was created.

This implementation also encountered issues with the compute tree when evaluating operations. Typically MagmaDNN will check if a tree has already been evaluated up to a certain point to avoid useful re-evaluation, but in the case of the first LSTM layer, it would re-evaluate every operation despite these checks, causing the program to be unusably slow on larger data sets. For these reasons, the second implementation of LSTM used one LSTM operation that performed all computations without calling other operations.

B. Second Approach Issues

The new implementation of the LSTM addresses many of the problems with the original implementation, notably the operation overhead and unnecessary revaluations of compute trees. However, this layer still has one notable problem: memory usage. When training for extended periods of time, such as on a large data set, the layer will eventually exhaust all available memory, causing a segmentation fault. The reason for this is not certain, but it is likely either a significant memory leak in the LSTM operation or an issue with how MagmaDNN's model.fit() method trains large numbers of weights (since the LSTM has 12 weights whereas other implemented layers will typically have 2). Another issue is that, in the current implementation of the LSTM, the bias matrix learns different biases for each batch when each batch should instead have the same vector of biases. That is, the biases are currently of shape batch_size by num_nodes, when they should be of shape num_nodes and use broadcasting to add with Tensor objects of shape batch_size by num_nodes.

VIII. Acknowledgment

This project was sponsored by the National Science Foundation through the Research Experience for Undergraduates (REU) award no. 2020534 with additional support from the National Institute of Computational Sciences and Innovative Computing Laboratory at the University of Tennessee, Knoxville. Dr. Kwai Wong was the mentor for this work.

References

[1] D. Nichols, K. Wong, S. Tomov, L. Ng, S. Chen, and A. Gessinger, "Magmadnn: Accelerated deep learning using magma," 2019.

[2] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[3] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid gpu accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5, pp. 232–240, 2010, parallel Matrix Algorithms and Applications. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819109001276

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[5] F. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: continual prediction with lstm," in *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, vol. 2, 1999, pp. 850–855 vol.2.

[6] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/