Mapping Multiple LSTM models on FPGAs

Stefano Ribes, Pedro Trancoso, Ioannis Sourdis
Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
{ribes, ppedro, sourdis}@chalmers.se

Christos-Savvas Bouganis
Department of Electrical and Electronic Engineering
Imperial College London
London, United Kingdom
christos-savvas.bouganis@imperial.ac.uk

Abstract—**Recurrent Neural Networks (RNNs) and their more recent variant Long Short-Term Memory (LSTM) are utilised in a number of modern applications like Natural Language Processing and human action recognition, where capturing long-term dependencies on sequential and temporal data is required. However, their computational structure imposes a challenge when it comes to their efficient mapping on a computing device due to its memory-bounded nature. As recent approaches aim to capture longer dependencies through the utilisation of Hierarchical and Stacked RNN/LSTM models, *i.e.* models that utilise multiple LSTM models for prediction, meeting the desired application latency becomes even more challenging. This paper addresses the problem of mapping multiple LSTM models to a device by introducing a framework that alters their computational structure opening opportunities for co-optimising the memory requirements to the target architecture. Targeting an FPGA device, the proposed framework achieves $3\times$ to $5\times$ improved performance over state-of-the-art approaches for the same accuracy loss, opening the path for the deployment of high-performance systems for Hierarchical and Stacked LSTM models.**

I. Introduction and Motivation

The recent advances in Machine Learning, especially Deep Neural Networks (DNN), have reignited the interest of researchers and practitioners on Neural Networks and their variations. With the abundant availability of data and computational capacity provided by modern GPUs, training of large DNNs with good generalisation properties became possible and led to unprecedented performance. Systems that rely on DNNs can efficiently perform a variety of tasks in applications from computer vision, to image understanding, scene analysis [1] and Natural Language Processing (NLP) [2].

In the case where long-term dependency capture is desired on sequential and temporal data, such as in the case of image captioning and NLP, Recurrent Neural Networks (RNN), a form of Neural Network with feedback connections, have demonstrated to be a suitable and efficient solution. However, standard RNNs suffer from vanishing and exploding gradients making their training a challenging task. An RNN variant, the Long-Short Term Memory (LSTM) network [3], addresses the above problem by introducing new structures, leading to their quick adoption in a large number of applications.

In case where the latency or throughput of the developed system is of concern, the mapping of LSTMs to a computing device is a challenging task, due to the low computation to communication ratio and the inherent dependencies in the LSTM operations. An LSTM network is based internally on structures (*i.e.* gates) that resemble networks with fully connected layers, that are manifested through matrix-vector multiplication operations, followed by non-linear functions. Any exploited parallelism is limited by the computational dependencies of the LSTM structure (*i.e.* recurrent connection). Moreover, the above problem is further amplified when multiple LSTMs need to be deployed as part of the application in the form of independent parallel executed LSTMs. Such case is where a number of independent outcomes are required based on the same input data, or in the utilisation of Stacked LSTMs [4], that extend the capabilities of LSTMs to longer time intervals. Example of applications can be found in [5] and [6], where Hierarchical Recurrent Neural Networks for skeleton-based action recognition are proposed, as well as in [7] where the authors propose a framework that utilises a two-stream Recurrent Neural Network pipeline for the task of action recognition. Finally, Li *et. al.* [8] propose a hierarchical LSTM model for building coherent long text for natural language generation and summarization.

Research effort in the efficient mapping of an LSTM to a device has focused only on the case where a single LSTM is required to be executed at any point of time [9], [10]. State-of-the-art approaches aim to increase the computation to communication ratio by reducing the memory accesses and computation cost through the investigation of parameter quantization and compression, as well as by pruning of connections (*i.e.* removing redundant network parameters [11]). Towards the above effort, the existing space can be divided into methods that require a re-training stage, allowing the methodologies to produce highly optimised designs [10], and approaches such as in [9] that assume no availability of data for retraining, focusing more on the generality of the approach.

This paper departs from the previously published approaches by focusing on the problem of mapping multiple LSTMs in a device, and more specifically in the case where these LSTMs are independent of each other apart except that they are part of the same application. Also, focusing on the generality of the approach, no assumption on the availability of training data is made.

The main contributions of the paper are as follows:

- a methodology is proposed for approximating for the

This work was supported by the European Commission under the Horizon 2020 Program through the ECOSCALE project (contract no. 671632) and the Swedish Research Council under the ScalaNetS project (2016-05231).

first time multiple LSTMs together, rather than each separately; the methodology allows iterative refinement of the LSTMs approximation leading to tunable and improved computation to communication ratios.

- an approach that exposes the computational and memory capabilities of the targeted device to the approximation algorithm, through structured pruning over the introduced refinement stages, leading to an architecture with improved device utilisation.

To the best of authors' knowledge this is the first work in the literature that addresses the important and timely problem of mapping multiple LSTMs on a device.

II. Background

A. LSTM Networks

An LSTM network processes an input $\mathbf{x}^t$ and produces an output $\mathbf{h}^t$ in every time-step t, where $\mathbf{x}$ and $\mathbf{h}$ denote vectors. Key to the operation of the LSTM is its recurrent connection of its output to its hidden units allowing the network to pass information over a number of time-steps, where regulation of the information flow is controlled through four modules called *gates*. Figure 1 illustrates the flow of an LSTM, where the details of the LSTM gates are given in Equation 1, where $\mathbf{b}$ and $\odot$ denote a bias vector and the element-wise multiplication operator. The *input* gate, $\mathbf{i}^t$, along with the *cell* gate $\mathbf{c}^t$ determine the amount of input information that propagates to the output of the network, whereas the *forget* gate, $\mathbf{f}^t$, controls the amount of previous information that will be maintained by the network. The *output* gate, $\mathbf{o}^t$, determines how much of the current state will be propagated to the network output.

The above gates are instantiated through non-linear functions, such as sigmoid $\sigma(\cdot)$ or hyperbolic tangent functions $tanh(\cdot)$, that operate on linear functions of the current input $\mathbf{x}^t$ and of the previous time-step output $\mathbf{h}^{t-1}$. Computationally, each gate is based on matrix-vector multiplications, and it is parameterised with a set of weight matrices, $\mathbf{W}_{cur}$ and $\mathbf{W}_{rec}$, responsible to modulate the current input and previous output.

$$\begin{aligned}
\mathbf{i}^t &= \sigma\big(\mathbf{x}^t \cdot \mathbf{W}_{cur_i} + \mathbf{h}^{t-1} \cdot \mathbf{W}_{rec_i} + \mathbf{b}_i\big)\\
\mathbf{f}^t &= \sigma\big(\mathbf{x}^t \cdot \mathbf{W}_{cur_f} + \mathbf{h}^{t-1} \cdot \mathbf{W}_{rec_f} + \mathbf{b}_f\big)\\
\mathbf{c}^t &= \mathbf{f}^t \odot \mathbf{c}^{t-1} + \mathbf{i}^t \odot \tanh\big(\mathbf{x}^t \cdot \mathbf{W}_{cur_c} + \mathbf{h}^{t-1} \cdot \mathbf{W}_{rec_c} + \mathbf{b}_c\big)\\
\mathbf{o}^t &= \sigma\big(\mathbf{x}^t \cdot \mathbf{W}_{cur_o} + \mathbf{h}^{t-1} \cdot \mathbf{W}_{rec_o} + \mathbf{b}_o\big)\\
\mathbf{h}^t &= \mathbf{o}^t \odot \tanh(\mathbf{c}^t)
\end{aligned} \tag{1}$$

B. SVD-Based Approximation

Typical DNNs, including LSTMs, utilise matrix-vector multiplication operations leading to designs whose performance is memory-bounded as a large number of parameters (*i.e.* weights matrix) needs to be accessed for the computation over a single input vector. Techniques to address this problem rely on batching multiple input vectors, sharing the weights access across multiple inputs, and/or pruning/approximating the weight matrices, reducing as such the data that need to be accessed per input. In the case of LSTMs, the former technique cannot be applied due to their recurrent connections, and effort is placed on the latter approach in order to improve the computation over communication ratio.

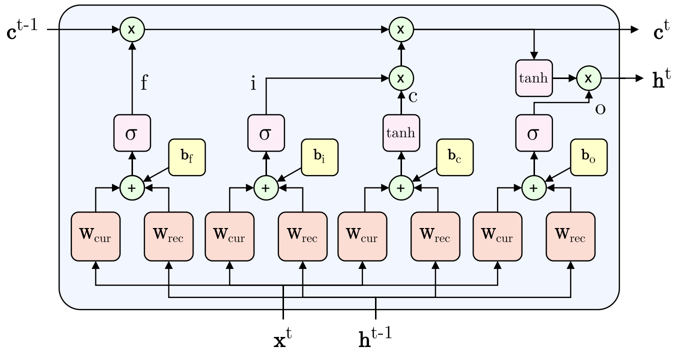


Fig. 1. LSTM flow for processing output $\mathbf{h}^t$ and cell state $\mathbf{c}^t$ at timestep t.

Possible techniques to prune/approximate the weights matrix include weights quantization, pruning of certain weights [10], as well as approximations of the weights matrix through rank-1 decomposition [12].

Decomposition of a matrix through rank-1 approximations expresses a matrix $\mathbf{W}$ as a linear combination of rank-1 matrices. The decomposition is achieved through the Singular Value Decomposition (SVD) algorithm that decomposes a given matrix $\mathbf{W}$ into 3 orthogonal matrices $\mathbf{U}$, $\mathbf{S}$, $\mathbf{V}$ as $\mathbf{W} = \mathbf{USV}^T$. The original matrix $\mathbf{W}$ can be approximated by selecting to utilise the first R rank-1 matrices of the decomposition (*i.e.* the ones that correspond to the largest eigenvalues), where the SVD algorithm guarantees the optimality of the approximation under the Mean Square Error (MSE) metric. As such, the matrix $\mathbf{W}$ can be approximated as:

$$\mathbf{W} \approx \sum_{i}^{R} s_i \ \mathbf{u}_i \ \mathbf{v}_i^T$$

where $\mathbf{u}_i$ and $\mathbf{v}_i$ correspond to the ith column and row of the $\mathbf{U}$ and $\mathbf{V}^T$ matrices respectively, while s_i is the ith element of the main diagonal of the diagonal matrix $\mathbf{S}$. The approximation leads to a reduction on the amount of data that need to be accessed as well as allows the matrix-vector multiplication computation to be performed through a series of dot-product calculations, as it will be shown later.

III. Problem Formulation

The work considers the general problem of accelerating the execution of multiple LSTM models that operate in parallel on synchronised inputs, and the device of choice is an FPGA. The parameters of the models are assumed to be stored in the off-chip memory, increasing the applicability of the approach to large problem sizes. The problem is formulated as follows: given a set of N LSTM models M_i with weight matrices $\mathbf{W}^i_{type}$, with $type \in \{cur_{gate}, rec_{gate}\}$ for $gate \in \{i, f, o, c\}$, and a target FPGA device D, derive an implementation that minimises the latency of their execution. More specifically, the work focuses on the case of a lossy mapping, where an error in the approximation on the final results of the computation is allowed but bounded by a user-specified threshold.

The proposed approach builds upon the work of Rizakis *et al.* [9], but it extends their problem formulation to address the case of multiple LSTMs. The key idea is to provide a decomposition of the weight matrices of the LSTMs in order to facilitate the necessary computations as a trade-off of latency and quality of the final result, along side with providing computational structures that would fully exploit the compute and memory capabilities of the targeted device.

Towards this, the proposed approach is based on the Singular Value Decomposition algorithm applied to a set of input matrices $\mathbf{W}_1, ..., \mathbf{W}_N$, producing a set of rank-1 matrices (*i.e.* matrices that can be expressed as the product of two vectors $\mathbf{u}^{(i)}$, $\mathbf{v}^{(i)}$) whose linear combination constructs the original input matrices. Such decomposition guarantees the least error in the approximation of the input matrices under the Mean Square Error (MSE) metric for a given number of rank-1 matrices used in the approximation [13].

As such, focusing on our problem formulation, the proposed approach aims to produce a *single set* of rank-1 matrices that approximates *all* the weight matrices of the same type $\mathbf{W}^i_{type}$ across the N LSTM targeted models. Thus, our approach allows us to *share* the $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ components across the N LSTM models M_j, and in doing so reduces the memory footprint of the models for a given targeted approximation error. Equation 2 indicates the approximation of a single matrix with R rank-1 matrices, where the $type$ and $gate$ indices have been dropped for clarity.

$$\mathbf{W}_{\mathrm{M}_j} \approx \sum_{i=1}^{R} {s_j}^{(i)} \odot \left(\mathbf{u}^{(i)} \cdot {\mathbf{v}^{(i)}}^T\right),\ j = 1, ..., N \qquad (2)$$

Algorithm 1 lists the necessary steps for decomposing N given weight matrices $\mathbf{W}_1, ..., \mathbf{W}_N$ into the R components $\mathbf{u}^{(i)}$, ${s_j}^{(i)}$ and $\mathbf{v}^{(i)}$. The algorithm also sparsifies and quantizes such components to improve the mapping to the device.

The algorithm begins by initializing a set of error matrices $\mathbf{E}_1, ..., \mathbf{E}_N$ and one set of approximated weight matrixes $\widetilde{\mathbf{W}}_1, ..., \widetilde{\mathbf{W}}_N$. After initialization, for each of the refinement steps R, the algorithm first updates the error matrices by taking the difference between the original matrices and the partially reconstructed ones, *i.e.* approximated (line 5). Upon constructing the new error matrices, at line 6 we apply the decomposition described in [13] to obtain the $\mathbf{u}^{(i)}$, ${s_j}^{(i)}$ and $\mathbf{v}^{(i)}$ components. This decomposition aims to minimize the MSE of the approximated matrices reconstructed from the $\mathbf{u}^{(i)}$, ${s_j}^{(i)}$ and $\mathbf{v}^{(i)}$ elements.

It has been shown in the literature that neural networks are able to maintain their accuracy after the sparsification of their weight matrices, *i.e.* setting most of their weight values to zero, thanks to a process called pruning [11]. A standard defacto way of pruning a network consists of an iterative process where a first step applies zero masks to the network matrices, followed by a fine-tuning step, *i.e.* retraining process. In this work, we propose a structured pruning of the $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ vectors that does *not* require retraining the network. Please note that ${s_j}^{(i)}$ are scalars and therefore are not pruned. In order to prune, we first divide the vectors $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ into T_u and T_v tiles respectively. Afterwards, we select the ZT_u tiles from vector $\mathbf{u}^{(i)}$ and the ZT_v tiles from $\mathbf{v}^{(i)}$ that contain the values with the minimum absolute magnitude, lines 8 and 12. Finally, all the elements of the selected tiles are assigned to zero. Pruning reduces both the amount of operations needed and the number of tiles, *i.e.* weight values, to be accessed.

Algorithm 1: Decomposition algorithm.

```
Data: $N \times \mathbf{W}$ weight matrices, $R$ number of refinement steps, $T_u$ and
      $T_v$ number of tiles, $ZT_u$ and $ZT_v$ number of tiles to prune.
1  begin
2      $\mathbf{E}_i \longleftarrow \mathbf{0}, \quad i = 1, ..., N$
3      $\widetilde{\mathbf{W}}_i \longleftarrow \mathbf{0}, \quad i = 1, ..., N$
4      for $i$ in $R$ do
5          $\mathbf{E}_j \longleftarrow \mathbf{W}_j - \widetilde{\mathbf{W}}_j, \quad j = 1, ..., N$
6          $\mathbf{u}^{(i)}, s^{(i)}, \mathbf{v}^{(i)} \longleftarrow$ decompose($\mathbf{E}$)
7          for $j$ in $ZT_u$ do
8              $zu^{(i)}[j] \longleftarrow \arg\min_k\{|\mathrm{mean}(\mathbf{u}^{(i)}[k])|\}$
9              $\mathbf{u}^{(i)}[zu^{(i)}[j]] \longleftarrow \mathbf{0}$            // pruning
10         end
11         for $j$ in $ZT_v$ do
12             $zv^{(i)}[j] \longleftarrow \arg\min_k\{|\mathrm{mean}(\mathbf{v}^{(i)}[k])|\}$
13             $\mathbf{v}^{(i)}[zv^{(i)}[j]] \longleftarrow \mathbf{0}$            // pruning
14         end
15         $\mathbf{A} \longleftarrow \mathbb{Q}(\mathbf{u}^{(i)}) \cdot \mathbb{Q}({\mathbf{v}^{(i)}}^T)$
16         $\widetilde{\mathbf{W}}_j \longleftarrow \mathbb{Q}(\widetilde{\mathbf{W}}_j) + \mathbb{Q}(s_j^{(i)}) \odot \mathbf{A},\ j = 1, ..., N$
17     end
18 end
```

Furthermore, a quantization operation of the pruned vectors $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ and the scalars ${s_j}^{(i)}$ (indicated with the $\mathbb{Q}(\cdot)$ operator) is performed, before the algorithm moves to the next refinement step. The quantized vectors are multiplied together to form a shared matrix $\mathbf{A}$ (line 15). The matrix $\mathbf{A}$ is then multiplied by the quantized scalars ${s_j}^{(i)}$ and added to the partial approximation matrix $\widetilde{\mathbf{W}}_j$.

At the next iteration, the approximation matrices $\widetilde{\mathbf{W}}_1, ..., \widetilde{\mathbf{W}}_N$ will include the errors introduced by both the pruning and quatization processes. The decomposition step will therefore generate components $\mathbf{u}^{(i)}$, $s^{(i)}$ and $\mathbf{v}^{(i)}$ that account for such errors, minimizing the overall MSE.

IV. Accelerator Design

In order to accelerate the execution of N LSTM layers, we applied our approximation algorithm to the weight matrices $\mathbf{W}_{type}$, with $type \in \{cur_{gate}, rec_{gate}\}$ and $gate \in \{i, f, c, o\}$. In particular, we approximated the weight matrices of the same type and gate together, because we empirically found them to have similar structures. For example, we made all the current forget gates ($\mathbf{W}^{cur}_f$) of the N LSTMs share the same $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ vectors (we will refer to this operation as *merging*). Merging same type of gate matrices overall yields lower MSE compared to stacking the gate weight matrices together and then approximating them. We believe that the reasons for this are twofold: first, the size of the approximated matrices is smaller, therefore the MSE can decrease quickly with fewer

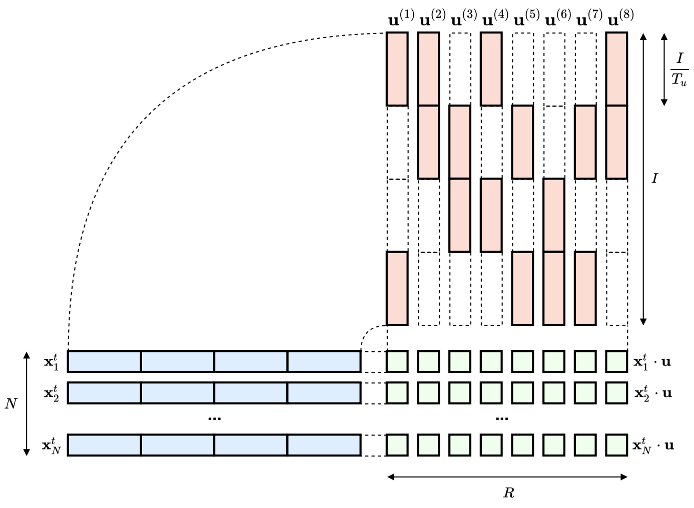

$\mathbf{u}^{(1)}$ $\mathbf{u}^{(2)}$ $\mathbf{u}^{(3)}$ $\mathbf{u}^{(4)}$ $\mathbf{u}^{(5)}$ $\mathbf{u}^{(6)}$ $\mathbf{u}^{(7)}$ $\mathbf{u}^{(8)}$
$\frac{I}{T_u}$
I
$\mathbf{x}_1^t$
$\mathbf{x}_2^t$
$\mathbf{x}_N^t$
N
$\mathbf{x}_1^t \cdot \mathbf{u}$
$\mathbf{x}_2^t \cdot \mathbf{u}$
$\mathbf{x}_N^t \cdot \mathbf{u}$
R

Fig. 2. Reduce products $\mathbf{x}_j^t \cdot \mathbf{u}^{(i)}$ with $R = 8$, $T_u = 4$ and $ZT_u = 2$.

refinement steps R. Second, the gates across different LSTM models perform a similar function and so the information filtered out by our algorithm tends to be the same, thus improving the approximation MSE. Nevertheless, the above behaviour is application dependent and other constructions should be considered.

Our FPGA accelerator's key computation is the approximated vector-matrix multiplication of the N LSTM inputs with the gate weight matrices, as exemplified in Equation 3, which approximates the multiplication between the input vectors $\mathbf{x}_j^t$ with the current forget gates weight matrices $\mathbf{W}_f^{cur}$.

$$\mathbf{x}_j^t \cdot \mathbf{W}_{f_j}^{cur} \approx \sum_{i=1}^{R} \left(\mathbf{x}_j^t \cdot \mathbf{u}_f^{(i)} \right) \cdot {s_f}_j^{(i)} \Big) \odot \mathbf{v}_f^{(i)}, \; j = 1, ..., N \quad (3)$$

The matrix-vector multiplication is effectively decomposed in three parts: a dot product ($\mathbf{x}_j^t \cdot \mathbf{u}^{(i)}$), a scalar-scalar multiplication ($\cdot {s_f}_j^{(i)}$) and finally a scalar-vector multiplication ($\odot \mathbf{v}_f^{(i)}$). Equation 3 is applied to both the current and the recurrent gates of the LSTMs, just by using, for the recurrent gates, the previous output $\mathbf{h}_j^{t-1}$ and the properly sized vector components. Notice that all the elements of the equation are quantized and that the vectors $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ are also pruned. A visual example of the R dot products of Equation 3 is depicted in Figure 2. In this example all the $\mathbf{u}^{(i)}$ vectors contain exactly two non-pruned tiles and two pruned tiles. The pruned tiles of the $\mathbf{u}^{(i)}$ vectors are dashed. Only the non-pruned tiles participate to the final computation, thereby saving time and resources required to perform the product.

The accelerator is designed in a dataflow fashion, illustrated in Figure 3. Inputs and weights are stored in the DRAM external memory and fed to the FPGA accelerator through the four available high performance AXI ports which are directly connected to the memory controller. Each AXI port is connected to a Direct Memory Access (DMA) unit that feeds the processing kernels with the respective data. The accelerator is composed of the following main building blocks:

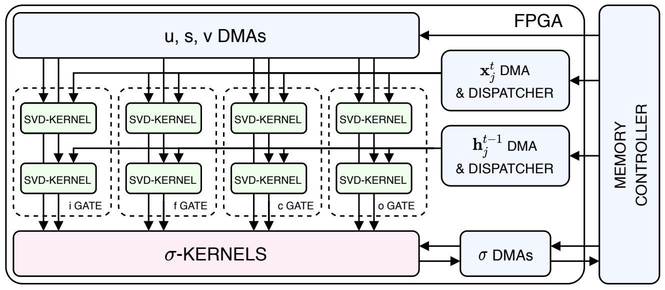


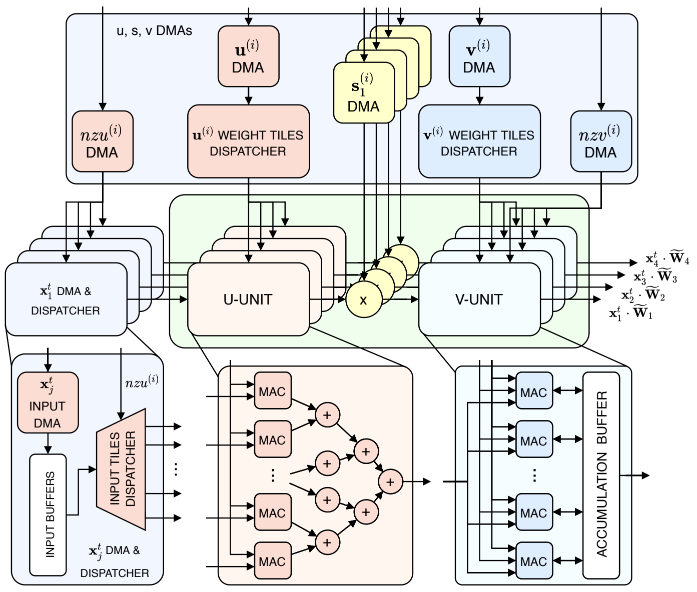
Fig. 3. The proposed dataflow accelerator architecture. The approximated current and recurrent LSTMs gates are processed in parallel by eight SVD-kernels. The σ-kernels compute the final steps of the LSTMs algorithm. The DMAs stream the required inputs and weights from memory to the kernels.

a) *SVD-Kernels*: they are responsible of the execution of the approximated matrix-vector operation of the LSTM gates, as reported in Equation 3. There are a total of 8 kernels, 4 for the current matrices of the LSTM gates and 4 for the recurrent ones.
b) *Input DMAs* and *tiles dispatchers*: they are in charge of transferring the inputs of the two LSTMs from main memory to the correct engines. In addition, they offer temporary on-chip buffers to store the N LSTMs' inputs $\mathbf{x}_j^t$ and $\mathbf{h}_j^{t-1}$ maximizing data reuse. Only the tiles corresponding to the non-pruned u-vector tiles are then read from the buffers and broadcasted into the MAC units of the SVD-kernels.
c) *u, s, v DMAs*: these DMA units fetch the non-zeroed tiles of the $\mathbf{u}^{(i)}$, $\mathbf{v}^{(i)}$, ${s_j}^{(i)}$ weight vectors to be streamed into the SVD-kernels.
d) *σ-Kernels*: their task is to apply the gate biases and the required non linear operations, listed in Equation 1, to the product of the inputs with the approximated weight matrices. There are N σ-kernels, one for each LSTM.
e) *σ DMAs*: these DMAs supply the data to the σ-kernels, *i.e.* the bias vectors and the previous LSTMs cell states. They also are used to write back the final computation to main memory.

The block diagram of the input DMA, tiles dispatchers and SVD-kernel is shown in Figure 4. The SVD-kernel computes Equation 3 and is composed of two types of units: U-unit and V-unit. Within the kernel, there are N U-units and N V-units. The U-units are responsible for computing the dot product reported in Equation 4.

$$\begin{gathered} {x_u}_j^{(i)} = \mathbf{x}_j^t[nzu_k^{(i)}] \cdot \mathbf{u}^{(i)}[nzu_k^{(i)}], \\ j = 1, ..., N; \quad k = 1, ..., T_u - ZT_u \end{gathered} \quad (4)$$

Each U-unit includes $T_u - ZT_u$ parallel multiply-accumulate blocks and an adder tree. In order for the U-units to perform their computation, the N input tiles dispatcher supply the non-pruned input tiles, while the $\mathbf{u}^{(i)}$ tile dispatcher broadcasts the non-pruned tiles. Thanks to the list of indexes nzu the N input tiles dispatchers read the input tiles corresponding to the non-pruned tiles of $\mathbf{u}^{(i)}$ and then stream them from their on-chip buffers to the respective MACs within the corresponding U-unit (recall Figure 2).



u, s, v DMAs
$\mathbf{u}^{(i)}$ DMA
$\mathbf{s}_1^{(i)}$ DMA
$\mathbf{v}^{(i)}$ DMA
$nzu^{(i)}$ DMA
$\mathbf{u}^{(i)}$ WEIGHT TILES DISPATCHER
$\mathbf{v}^{(i)}$ WEIGHT TILES DISPATCHER
$nzv^{(i)}$ DMA
$\mathbf{x}_1^t$ DMA & DISPATCHER
U-UNIT
x
V-UNIT
$\mathbf{x}_4^t \cdot \widetilde{\mathbf{W}}_4$
$\mathbf{x}_3^t \cdot \widetilde{\mathbf{W}}_3$
$\mathbf{x}_2^t \cdot \widetilde{\mathbf{W}}_2$
$\mathbf{x}_1^t \cdot \widetilde{\mathbf{W}}_1$
$\mathbf{x}_j^t$ INPUT DMA
$nzu^{(i)}$
INPUT BUFFERS
INPUT TILES DISPATCHER
$\mathbf{x}_j^t$ DMA & DISPATCHER
MAC
+
ACCUMULATION BUFFER

Fig. 4. Dataflow architecture of one of the eight SVD-kernels for processing four LSTMs. The kernel is composed of two types of sub-blocks, the U-Unit and the V-Unit. There is one set of U-Unit and V-Unit per input. All U-Units (V-Units) are fed by the *same* $\mathbf{u}^{(i)}$ DMA and $\mathbf{u}^{(i)}$ weight tiles dispatcher ($\mathbf{v}^{(i)}$ DMA and $\mathbf{v}^{(i)}$ weight tiles dispatcher), since the $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ vectors are *shared* across the different LSTMs.

The $N \times R$ scalars $x_{u_j}^{(i)}$ produced by the U-units are then multiplied by the $s_1^{(i)}, ..., s_j^{(i)}$ scalar components and forwarded to the kernel's V-units as $x_{s_j}^{(i)}$. The V-units perform the operations in Equation 5, *i.e.* the last step of the approximation process.

$$\mathbf{x}_j^t \cdot \widetilde{\mathbf{W}}_j \approx \sum_{i=1}^{R} x_{s_j}^{(i)} \odot \mathbf{v}^{(i)}[nzv_k^{(i)}] \qquad (5)$$
$$j = 1, ..., N; \quad k = 1, ..., T_v - ZT_v$$

Like for the U-units, there is a weight dispatcher which is in charge of supplying the V-unit's MACs with the non-pruned $\mathbf{v}^{(i)}$ vector tiles. In order to multiply and accumulate the $x_{s_j}^{(i)}$ scalars with the non-pruned $\mathbf{v}^{(i)}$ weight elements, each V-unit utilizes a partitioned accumulation buffer. The buffer is partitioned tile-wise to allow parallel access to it from the MACs. Once the refinement steps are completed, the V-units stream out the final approximated products $\mathbf{x}_j^t \cdot \tilde{\mathbf{W}}_j$ from their accumulation buffers.

Finally, the results of the SVD-kernels are streamed to the σ-kernels for applying the last non-linear functions required by the LSTMs.

V. Proposed Framework

In this section we describe a framework for identifying the combination of design parameters which best tradeoff the accuracy and execution time of accelerating N LSTM models. The initial part of the section describes our methodology, while the next and final one details the roofline model we use to estimate the performance of our accelerator during the design space exploration phase.

TABLE I
List of design parameters of the framework.

Symbol	Description
R	Amount of refinement steps.
T_u	Number of tiles of the $\mathbf{u}^{(i)}$ vectors.
ZT_u	Number of pruned tiles of the $\mathbf{u}^{(i)}$ vectors.
T_v	Number of tiles of the $\mathbf{v}^{(i)}$ vectors.
ZT_v	Number of pruned tiles of the $\mathbf{v}^{(i)}$ vectors.
B	Byte size of the quantized LSTM's input and weight values.

A. Methodology

There is a large number of design parameters, *i.e.* number of refinement steps, tile size, pruning percentage and quantization (detailed in Table I), each having a large range of possible options, which make the design space huge and impractical to search exhaustively. We have defined the following methodology to select one, or at most a few, design points, which are promising for achieving a good performance-accuracy trade-off and fit in the target FPGA device.

First, we set particular performance and accuracy goals, as well as the resource constraints for our target design. Designs with accuracy below a certain threshold or excessive need for resources are not further considered. However, measuring actual accuracy requires heavy application-level simulation of the particular design point. Similarly, measuring actual resource requirements and performance requires a design implementation, which is time consuming. Our experiments indicate that the most critical and limited device resources are the DSP slices and BRAMs, for which analytical models have been developed. The proposed approach adopts analytical models that provide indications for accuracy, need for critical resources, as well as for performance for each design point. Based on these models we select the most promising design points for further evaluation and eventually implementation.

Second, we search the design space based on the accuracy criterion. We get an indication of the accuracy drop compared to the original application (before SVD approximation, pruning and quantization, etc.) using the average Mean Square Error (MSE) between the original stacked weight matrices $\mathbf{W}$ and the SVD approximated ones $\widetilde{\mathbf{W}}$, defined in Equation 6. Subtraction and square operations are performed element-wise.

$$\text{MSE}(\mathbf{W}, \widetilde{\mathbf{W}}) = \text{mean}\big((\mathbf{W} - \widetilde{\mathbf{W}})^2\big) \qquad (6)$$

Our conjecture, confirmed in the next section, is that a small MSE is a necessary condition for low accuracy drop. Consequently, design points with MSE below a MSE threshold (T_{MSE}) are selected for further evaluation. These design points are subsequently selected for simulation in order to measure their actual accuracy drop. Out of those, the design points with actual accuracy drop below our accuracy threshold (T_{acc}) are selected to continue in the next step of our design space exploration process.

Third, the design points that passed the accuracy check are then evaluated for their resource requirements and performance. In order to avoid generating a hardware implementation for all of them, the need for DSP slices and

BRAMs is estimated. Designs that need more critical resources than available on the device are dropped. Subsequently, the attainable performance based on our roofline model (described in the following subsection) is used to estimate their execution time as in Equation 7.

$$t_{exe} = \frac{Nops}{Att_{perf}} \left[\frac{Ops}{Ops/s} \right] \tag{7}$$

The designs with the lowest attainable execution time are finally implemented in the FPGA board at hand and their actual performance is measured.

B. Roofline Model

For estimating the execution time of our FPGA accelerator we derive a roofline model for calculating the attainable performance of the possible designs [14], [15]. The attainable performance is defined in Equation 8 as the minimum value between the Computational Performance (CP) and the product between the maximum available bandwidth of the system B_w and the Communication To Computation ratio (CTC).

$$Att_{perf} = \min \{ CP,\ CTC \cdot B_w \} \left[\frac{Ops}{s} \right] \tag{8}$$

The CP can be estimated as in Equation 9, where N_{ops} and N_{cycles} are the total number of performed fixed point operations and the estimated amount of execution cycles, respectively.

$$CP = \frac{Nops}{Ncycles \cdot \frac{1}{f_{clk}}} \left[\frac{Ops}{s} \right] \tag{9}$$

For our accelerator, the amount of required operations is reported in Equation 10. In an LSTM there are four gates, each including a pair of current and recurrent matrices, giving 8 matrices in total, four of which having dimension $I \times H$ and four $H \times H$. The U-units and V-units perform a series of MAC operations, so 1 MAC corresponds to two operations. The amount of non-linear operations on each hidden value is estimated to be equal to 24, leading to $Nops_\sigma$ amount of operations for the σ-kernel.

$$\begin{aligned}
Nops &= N \cdot (Nops_u + Nops_s + Nops_v + Nops_\sigma) \\
&= N \cdot (Nops_u + R \cdot 8 + Nops_v + 24 \cdot H) \\
Nops_u &= R \cdot \Big(4 \cdot (T_u - ZT_u) \cdot \Big(\frac{I}{T_u} + \frac{H}{T_u} \Big) \Big) \cdot 2 \\
Nops_v &= R \cdot 8 \cdot (T_v - ZT_v) \cdot \frac{H}{T_v} \cdot 2
\end{aligned} \tag{10}$$

In order to finally compute the CP value, we need to estimate the required execution cycles, *i.e.* the accelerator's latency. The accelerator's latency is reported in Equation 11. Since the accelerator is designed in a dataflow fashion, we only consider the slowest accelerator's module and therefore the overall latency will be the maximum latency value among the hardware modules. Please notice that each LSTM weight matrix is mapped to a different SVD-kernel, so there are 8 SVD-kernels in total running in parallel. The N inputs, corresponding to N LSTM models, are also processed in parallel within each SVD-kernel.

$$\begin{aligned}
Ncycles &= \max \{ U_{latency}, S_{latency}, V_{latency}, \sigma_{latency} \} \\
&= \max \Big\{ U_{latency},\ R,\ R\ (T_v - ZT_v),\ 7\ \frac{H}{T_v} \Big\} \\
U_{latency} &= R\ \max \Big\{ \frac{I}{T_u},\ \frac{H}{T_u},\ log_2(T_u - ZT_u) \Big\}
\end{aligned} \tag{11}$$

The last value we need for calculating the attainable performance is the CTC, which is reported in Equation 12. The CTC is defined as the ratio between the total number of operations $Nops$ in Equation 10 and the total amount of transferred data (in Bytes), reported in Equation 13.

$$CTC = \frac{Nops}{in + out + w + nz + bias} \left[\frac{Ops}{Byte} \right] \tag{12}$$

$$\begin{aligned}
in + out &= N \cdot \Big((I + H) + 2\ H \Big) \cdot B \\
nz + bias &= R \cdot 8 \cdot (T_u + T_v)/8 + 4 \cdot H \cdot N \cdot B \\
w &= u_{size} + s_{size} + v_{size} \\
u_{size} &= R \cdot 4 \cdot (T_u - ZT_u) \cdot \Big(\frac{I}{T_u} + \frac{H}{T_u} \Big) \cdot B \\
s_{size} &= R \cdot 8 \cdot N \cdot B \\
v_{size} &= R \cdot 8 \cdot (T_v - ZT_v) \cdot H/T_v \cdot B
\end{aligned} \tag{13}$$

The values that need to be read and written are divided in several groups. The in and out values comprise the input and output vectors for the N LSTM models. The weights that the accelerator requires are the $bias$ values, the non-zero indexes nz (which are bit vectors of size proportional to the amount of tiles T_u and T_v) and finally the approximated weight values w. The value of w includes the u, s and v components, which sizes are referred to as u_{size}, s_{size} and v_{size}.

VI. Evaluation

In this section we describe the experimental setup and present a validation of the models described in the previous section followed by the evaluation of the proposed design in terms of performance and obtained accuracy.

A. Experimental setup

For the evaluation of the proposed framework, a multiple LSTM model that is trained for the Fashion MNIST dataset [16] is utilised. The Fashion MNIST dataset is a drop-in substitute for the MNIST dataset, but the classification task is considered more challenging [17], [18]. The targeted network model consists of two main branches, each containing an LSTM model [19]. For performance results, the software runs on the Processing System (PS) of the FPGA, which features four Cortex-A53 MPCore processors, ARMv8 architecture, running at 1.2GHz. The accuracy was tested by plugging in our HLS implementation to the Keras execution flow. The neural network was modeled in Keras 2.2.4 using Tensorflow 1.13.1 as a back-end.

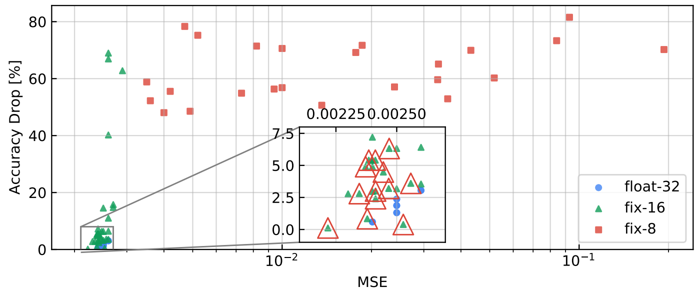


Fig. 5. Correlation between accuracy drop and MSE of the approximation.

For the evaluation of the proposed hardware architecture (denoted as *SVDn-HW*), we used a Xilinx Zynq UltraScale+ MPSoC ZCU104 FPGA. In order to generate the description of the hardware module from its high-level representation, we used Xilinx SDSoC 2018.3 tool.

We compared our proposed system against two software and two hardware implementations:

- *LSTM-SW*: Software implementation of baseline LSTM models using GEMV function from OpenBLAS library. Float32 values are used for both activations and weights.
- *LSTM-HW*: Hardware (FPGA) implementation of baseline LSTM models comprised of 8 parallel 1D systolic arrays for the dense matrix-vector computation (loosely inspired by [20]), followed by a non-linear unit.
- *SVDn-SW*: Software implementation of the SVD optimization of the LSTM models that utilizes the same weight values of *SVDn-HW* before quantization. *SVDn-SW* performs computations on dense weight matrices, despite having many zero values since the OpenBLAS library does not support sparse computation.
- *SVD1-HW*: A hardware (FPGA) implementation following the design methodology described in [9], where the mapping of each LSTM model is optimised in isolation.

B. Validation of accuracy, performance and resource models

Next, we present a brief validation of the accuracy, performance and resource models presented in Section V.

Regarding the validation of the accuracy model, in Figure 5 we show different design points for the proposed architecture characterized by the (average) MSE of its approximated LSTM weight matrices and the accuracy drop of the result, when compared to the correct output. Note that in this Figure we include design points for 32-bit floating-point as well as 8- and 16-bit fixed point implementations.

In general, it is possible to observe that the lower accuracy drop occurs for the lower values of MSE. Nevertheless, there are design points where a low MSE results in high accuracy drop. Consequently, choosing a design point with low MSE is a necessary but not sufficient condition for achieving a low accuracy drop of the result. An in-depth view is shown with the expansion of the bottom left corner, where it is possible to observe a correlation between the MSE and the accuracy drop. The values in that region are though very small, with very small differences between themselves. The red triangles

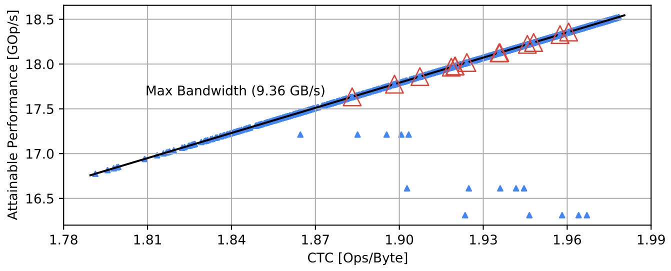


Fig. 6. Roofline model for our FPGA accelerator processing two LSTM layers with $I = 1024$ and $H = 512$.

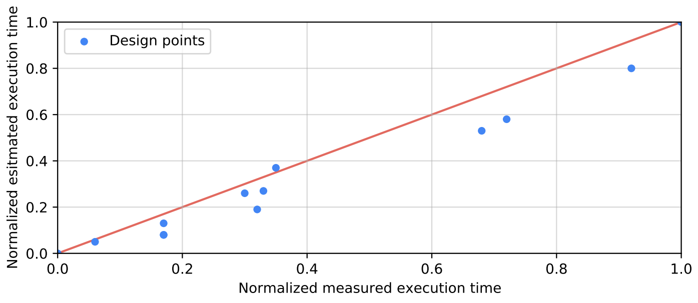


Fig. 7. Normalized observed execution time versus estimated execution time.

in the expanded section show the design points that we have selected to explore in more detail.

Figure 6 shows the roofline model of our accelerator processing two LSTM layers. The design points in the roofline have different parametrizations of the elements reported in Table I. We can notice that most of the design points hit the bandwidth limit, meaning that the computation is mainly memory-bound. The points highlighted by red triangles are the ones selected in Figure 5 based on accuracy. Based on the attainable performance of the roofline model, the validation of the execution time estimation model (in equation 7) is depicted in Figure 7. The estimated and measured values of execution time are normalized to each one's corresponding largest value. From this Figure it is possible to observe a high correlation between the estimated and the measured execution time, thus allowing us to use the model as a way to predict which designs achieve higher performance.

Lastly we validated the model for hardware resources. The DSP utilization estimate perfectly matches the count reported in place and route. The estimated BRAM usage shows a 1% relative error on average when compared with HLS reports and 18% versus post place and route results. We believe that the high error compared to post place and route BRAM utilization is because the Xilinx SDSoC 2018.3 tool introduces (when available) additional BRAMs for optimizations, which are hard to foresee and accurately estimate.

C. Evaluation of the proposed design

Next, the evaluation of the proposed design is presented in terms of execution time and accuracy drop of the output result and compare it to the alternative designs. The results are shown in Figure 8. The first observation is that, as expected,

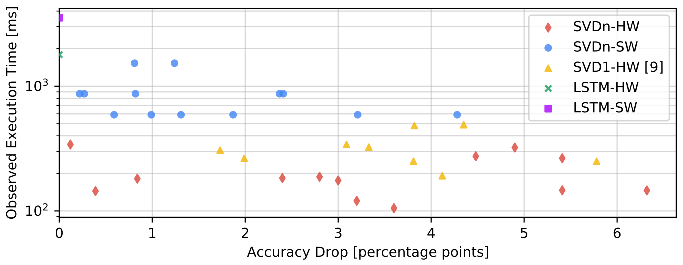


Fig. 8. Actual exec. time vs. accuracy drop. Orig. network accuracy is 84.4%.

the baseline implementations without approximation (*LSTM-SW* and *LSTM-HW*) are the only ones achieving a 0% accuracy drop. Nevertheless, this is achieved at a high latency, higher than any other design presented. Another expected observation is the fact that all *SVDn-SW* points have a higher latency than the corresponding *SVDn-HW* points. The difference observed ranges between a factor of 3.1× and 5.6×. Another interesting comparison is between the proposed *SVDn-HW* and the previously proposed *SVD1-HW*. In particular, it can be observed that the fastest *SVDn-HW* design is 1.7× faster than the fastest *SVD1-HW*, considering all plotted points have acceptable accuracy. The most accurate *SVDn-HW* design has 14x lower accuracy drop than the most accurate *SVD1-HW*, considering all plotted points have acceptable performance. This is explained by the fact that *SVD1-HW* applies a similar SVD-based methodology as our approach but does not exploit possible redundancies between weight matrices across LSTM models. As there is a trade-off between accuracy drop and performance, the best *SVDn-HW* design in the pareto-front is 2× faster and 4.5× more accurate than the best *SVD1-HW*.

VII. Related Work

Significant research effort has been focused on the efficient mapping of computationally heavy Convolutional Neural Networks on devices, leading to a number of automated toolchains [21] [22], [23] and compression methods [11], [24], [25]. In contrast to the CNN mapping, mapping of Recurrent Neural Networks and their variants (LSTMs) pose different challenges as the systems are memory-bounded.

As such, previous research aiming to address the memory-bound limitation of accelerating the execution of given LSTM models have focused in the reduction of either the data volume transferred between the off-chip memory and accelerator, or the amount of data that needs to be stored, thus enabling their complete storage on device. Early representative works in this area are [26], [27]. Common investigated techniques include parameter pruning, parameter sharing, and compression using lossy and lossless schemes. In parallel, effort has been put on the design of accelerator architectures that support the sparsity of the data and the computational patterns introduced by those compression methods [10], [11].

In the case where the LSTM optimisation can be considered during the training stage, research effort has focused on the extreme quantization of the parameters even to binary values [28]. However, the underlying assumption of availability of training data prohibits the application of those approaches in a large number of cases. Thus, effort has been placed on approaches that can be applied post-training. ESE [10] propose a load-balancing aware compression methodology, along-side an FPGA-specific architecture for speech processing. The compression scheme is based on parameter pruning and quantization, where their proposed architecture can operate directly with irregular patterns. To further address load balancing challenges stemmed from sparse parameter matrices, [29] and [30] propose novel sparse matrix formats, which allow improved load balancing capabilities across the processing elements. Nevertheless, even though the above methods do not require a training step, access to the training data is required for the pruning and the fine-tuning of the weights in order to achieve minimum penalty on the accuracy. Significant performance gains have been reported for custom hardware-based solutions in the case where the on-chip device memory can accommodate the parameters of the LSTM model, removing as such the requirement of accessing off-chip memory [28], [31]–[33]. However, such assumption severely restricts the application of these approaches and only few works [10], [26], [27], [34] address the general problem where the parameters of the compressed LSTM model do not fit in the on-chip memory, as is the case of the work presented in this paper.

Closer to this work are the works by Kouris *et. al.* [12] and Rizakis *et. al.* [9], that propose an SVD-based refining scheme for the approximation of the LSTM weight matrices.

The proposed work considers the more complex problem of mapping on a computing device multiple LSTM models that operate on synchronised inputs. The work focuses on exploiting any redundancies within and across the parameters of the models in order to produce a mapping that co-optimised the execution of all models. Previous pruning-based approaches can be used to further extend the impact of the proposed work through their application on each refinement stage, leading towards sparse computations, rather than aiming for a structured sparsity.

VIII. Conclusions

The paper presented a framework for the efficient mapping of multiple LSTMs on an FPGA device. By altering the structure of the computations it allows the co-optimisation of the scheduling of such computations and the underlying hardware parameters, while taking into account the resource constraints of the targeted device. The presented methodology offers the first compression scheme across multiple LSTM models. It offers better accuracy and performance compared to handling each LSTM separately and can be integrated with other existing lossy and lossless compression approaches. Even though a structured pruning approach is investigated in this work, the framework can be extended to allow a hybrid approach where each tile can be expressed through a sparse structure, allowing as such a finer design space exploration of the performance and computation to communication ratio.

References

[1] W. Byeon, T. M. Breuel, F. Raue, and M. Liwicki, "Scene labeling with LSTM recurrent neural networks," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 3547–3555.

[2] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur, "Recurrent neural network based language model," in *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, T. Kobayashi, K. Hirose, and S. Nakamura, Eds. ISCA, 2010, pp. 1045–1048. [Online]. Available: http://www.isca-speech.org/archive/interspeech_2010/i10_1045.html

[3] K. Greff, R. K. Srivastava, J. Koutnk, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[4] R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio, "How to construct deep recurrent neural networks," in *Proceedings of the Second International Conference on Learning Representations (ICLR 2014)*, 2014.

[5] Yong Du, W. Wang, and L. Wang, "Hierarchical recurrent neural network for skeleton based action recognition," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1110–1118.

[6] S. Zhang, X. Liu, and J. Xiao, "On geometric features for skeleton-based action recognition using multilayer lstm networks," in *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2017, pp. 148–157.

[7] H. Wang and L. Wang, "Modeling temporal dynamics and spatial configurations of actions using two-stream recurrent neural networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 3633–3642. [Online]. Available: https://doi.org/10.1109/CVPR.2017.387

[8] J. Li, M.-T. Luong, and D. Jurafsky, "A hierarchical neural autoencoder for paragraphs and documents," *arXiv preprint arXiv:1506.01057*, 2015.

[9] M. Rizakis, S. I. Venieris, A. Kouris, and C. Bouganis, "Approximate FPGA-Based LSTMs Under Computation Time Constraints," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications - 14th International Symposium, ARC 2018, Santorini, Greece, May 2-4, 2018, Proceedings*, 2018, pp. 3–15. [Online]. Available: https://doi.org/10.1007/978-3-319-78890-6_1

[10] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA 17. New York, NY, USA: Association for Computing Machinery, 2017, p. 7584. [Online]. Available: https://doi.org/10.1145/3020078.3021745

[11] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[12] A. Kouris, S. I. Venieris, M. Rizakis, and C.-S. Bouganis, "Approximate LSTMs for Time-Constrained Inference: Enabling Fast Reaction in Self-Driving Cars," *ArXiv*, vol. abs/1905.00689, 2019.

[13] C.-S. Bouganis, S.-B. Park, G. A. Constantinides, and P. Y. Cheung, "Synthesis and optimization of 2D filter designs for heterogeneous FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 1, no. 4, pp. 1–28, 2009.

[14] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[15] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.

[16] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms.

[17] "Fashion-MNIST," https://github.com/zalandoresearch/fashion-mnist, accessed: 2020-11-13.

[18] "Basic classification: Classify images of clothing," https://www.tensorflow.org/tutorials/keras/classification, accessed: 2020-11-13.

[19] "Hierarchical RNN (HRNN) to classify MNIST digits," https://github.com/keras-team/keras/blob/master/examples/mnist_hierarchical_rnn.py, accessed: 2020-06-08.

[20] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, "Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 244–254.

[21] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions," *ACM Comput. Surv.*, vol. 51, no. 3, Jun. 2018. [Online]. Available: https://doi.org/10.1145/3186332

[22] S. I. Venieris and C. Bouganis, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 40–47.

[23] ——, "fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 2, pp. 326–342, 2019.

[24] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. K. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware: Where weve been, where were going," *ACM Comput. Surv.*, vol. 52, no. 2, May 2019. [Online]. Available: https://doi.org/10.1145/3309551

[25] A. Kouris, S. I. Venieris, and C. Bouganis, "CascadeCNN: Pushing the Performance Limits of Quantisation in Convolutional Neural Networks," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 155–1557.

[26] A. X. M. Chang and E. Culurciello, "Hardware accelerators for recurrent neural networks on FPGA," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[27] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017, pp. 629–634.

[28] V. Rybalkin, A. Pappalardo, M. M. Ghaffar, G. Gambardella, N. Wehn, and M. Blott, "FINN-L: Library Extensions and Design Trade-Off Analysis for Variable Precision LSTM Networks on FPGAs," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 89–897.

[29] J. Park, W. Yi, D. Ahn, J. Kung, and J. Kim, "Balancing Computation Loads and Optimizing Input Vector Loading in LSTM Accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2019.

[30] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 6372. [Online]. Available: https://doi.org/10.1145/3289602.3293898

[31] J. C. Ferreira and J. Fonseca, "An FPGA implementation of a long short-term memory neural network," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2016, pp. 1–8.

[32] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung, "FPGA-Based Low-Power Speech Recognition with Recurrent Neural Networks," in *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, 2016, pp. 230–235.

[33] V. Rybalkin, N. Wehn, M. R. Yousefi, and D. Stricker, "Hardware architecture of Bidirectional Long Short-Term Memory Neural Network for Optical Character Recognition," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 1390–1395.

[34] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 152–159.