Streaming Overlay Architecture for Lightweight LSTM Computation on FPGA SoCs

LENOS IOANNOU, University of Warwick, UK
SUHAIB A. FAHMY, King Abdullah University of Science and Technology (KAUST), Saudi Arabia

Long-Short Term Memory (LSTM) networks, and **Recurrent Neural Networks (RNNs)** in general, have demonstrated their suitability in many time series data applications, especially in **Natural Language Processing (NLP)**. Computationally, LSTMs introduce dependencies on previous outputs in each layer that complicate their computation and the design of custom computing architectures, compared to traditional feed-forward networks. Most neural network acceleration work has focused on optimising the core matrix-vector operations on highly capable FPGAs in server environments. Research that considers the embedded domain has often been unsuitable for streaming inference, relying heavily on batch processing to achieve high throughput. Moreover, many existing accelerator architectures have not focused on fully exploiting the underlying FPGA architecture, resulting in designs that achieve lower operating frequencies than the theoretical maximum. This paper presents a flexible overlay architecture for LSTMs on FPGA SoCs that is built around a streaming dataflow arrangement, uses DSP block capabilities directly, and is tailored to keep parameters within the architecture while moving input data serially to mitigate external memory access overheads. The architecture is designed as an overlay that can be configured to implement alternative models or update model parameters at runtime. It achieves higher operating frequency and demonstrates higher performance than other lightweight LSTM accelerators, as demonstrated in an FPGA SoC implementation.

CCS Concepts: • **Computer systems organization** → **Neural networks**; **Reconfigurable computing**; *Embedded hardware;*

Additional Key Words and Phrases: LSTM, neural networks, overlay, machine learning

ACM Reference format:
Lenos Ioannou and Suhaib A. Fahmy. 2022. Streaming Overlay Architecture for Lightweight LSTM Computation on FPGA SoCs. *ACM Trans. Reconfig. Technol. Syst.* 16, 1, Article 8 (December 2022), 26 pages.
https://doi.org/10.1145/3543069

1 INTRODUCTION

Various **Neural Network (NN)** topologies have demonstrated good performance in specific domains, for example, **Convolutional Neural Networks (CNNs)** are widely used in computer vision while **Recurrent Neural Networks (RNNs)** work well for time-series data. Hybrid NN

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council (EPSRC) under grant EP/N509796/1 and in part by a Royal Academy of Engineering/The Leverhulme Trust Research Fellowship.
Authors' addresses: L. Ioannou, School of Engineering, Library Rd, Coventry CV4 7AL, United Kingdom; email: l.ioannou@warwick.ac.uk; S. A. Fahmy, Computer, Electrical and Mathematical Sciences and Engineering, King Abdullah University of Science and Technology (KAUST), Thuwal, 23955, Saudi Arabia; email: suhaib.fahmy@kaust.edu.sa.

1936-7406/2022/12-ART8
https://doi.org/10.1145/3543069

structures are also used for more complex tasks, for example, a **Long Short Term Memory (LSTM)** network, a type of RNN, can be used after a CNN to generate captions for images [1]. **Fully Connected (FC)**, or dense layers, are often embedded in the last parts of these networks to implement the classification or regression task. Alternatively, NNs comprising only dense layers can be used for less complex tasks such as specific event detection [2]. The increasing popularity of NNs has driven significant efforts to accelerate computation of different network topologies on a heterogeneous spectrum of computing platforms, from powerful servers to less capable devices at the edge.

NNs are typically trained on highly parallel GPU platforms due to the high computational workloads that suit offline centralised implementation. Inference scales well on more constrained devices since various optimisations can be applied [3–6]. Hence, there has been ample research on architectures for NN inference acceleration on a variety of platforms. A number of silicon vendors have also augmented processors with specialised neural processing units that offer the required parallelism, enabling significant acceleration of these workloads [7, 8].

Most previous work on FPGAs has focused on accelerating the generic matrix-vector operations used for NN inference. Weight pruning and quantization have also been widely used to effectively reduce the memory requirements of models. Other work has focused on bridging the gap between software programmable platforms and FPGAs by proposing automated toolflows [9]. In the same context, Xilinx Vitis enables compilation of accelerators from higher level standard frameworks. Much of the published work targets more capable FPGAs on servers, with high bandwidth PCIe interconnect [10–13]. Though research in the embedded domain has also flourished, many of these efforts either rely heavily on batch processing to generate high throughput, thus underperforming on single network inference and streaming data applications, or time multiplex complex compute units, thus not fully exploiting parallelism. Other optimisation methods include extreme quantization, even down to single bit data, and pruning. Although neural networks have been shown to tolerate such optimisations, these come at the cost of flexibility in the compute architecture while requiring additional design space and accuracy exploration, in addition to quantisation aware training. Finally, the majority of published work does not consider the FPGA architecture in detail, so fails to maximise achievable frequency. This results in lower performance than what should be achievable and poorer energy efficiency since leakage power is clock independent [10]. Some work on large scale matrix multiplication on datacenter FPGAs has demonstrated near theoretical maximum performance, but relies on large FPGA fabrics to enable full unrolling of NN computations [10, 11].

Many consumer applications rely on processing in both embedded and datacenter contexts. For example, the widely used voice assistants, such as Amazon Alexa and Apple Siri [14, 15], process natural language both at the edge device and in the datacenter. The edge device is responsible for wakeword detection, e.g., “Hey Siri”, through the use of lightweight NNs, while the words that follow are processed in the cloud. At the edge, hybrid processing is usually employed to maximise efficiency which includes the use of a low-power, always-on processor along with the device’s main processor. A lighter network runs on the low-power processor and once this network generates a value that exceeds a threshold, the main microprocessor is woken up to run a more complex network. Depending on the device’s capabilities, these networks can indicatively be five layers deep, using fully connected layers with either 32, 128, or 192 neurons [15]. Hence, we envisage a growing need for edge devices to accommodate lightweight or moderate sized NNs, to support offloading of further processing to the cloud. This could be a result of the constrained resources of the edge device, or to protect the intellectual property of an organisation by adding a layer between the edge device and their trained Neural Network. Architectures for this purpose should be self-contained and flexible enough to support different NN structures dynamically. Most previous

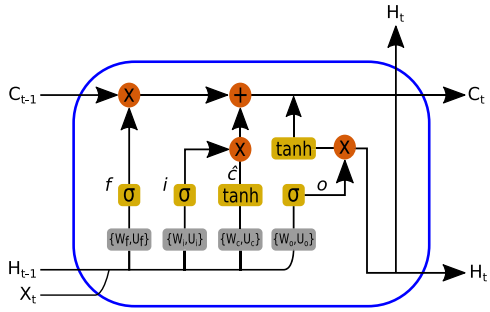


Fig. 1. An LSTM unit.

work has proposed co-processors that rely on a host processor to coordinate their operation and manage data transfers or only considered one layer type, offloading others to general purpose architectures.

LSTMs combined with fully connected layers are an ideal combination for processing time-series data in such lightweight applications, especially in wakeword or event detection, and hence we focus on these. Lightweight LSTM NNs have found applications in healthcare [16], weather prediction [17], and network security [18], among other applications. While fully connected layers are the most regular form of NNs, LSTMs include data feedback from previous timestep results and disrupt the regular flow of data, which in turn breaks up back to back matrix multiplications. Although this can be somewhat alleviated with batch processing or by executing multiple NNs simultaneously, both methods increase the volume of intermediate results to be cached, and may not fit all application data rate requirements.

In this work, we propose an overlay architecture that can process Fully Connected (FC) and LSTM layers flexibly, while operating at high frequency. Low level computations are abstracted to building blocks that can easily be replicated to reflect the structure of a model, while tailoring the datapath to their complex dataflow pattern. Support for the various activations functions is provided through approximations that maintain low complexity and resource utilization. The proposed architecture is self contained and flexibly reconfigurable to implement different models and adapt to weight updates. Our approach is tailored to operate within edge SoC environments and can be used to accommodate lightweight to moderate NN workloads. It operates in streaming mode, with computations carefully mapped to DSP blocks, each mimicking the operation of a neuron, leaving LUTs for weight storage and other functionality. This architecture caches very few intermediate results as they are consumed by subsequent processing units in a streaming manner. Finally, the overlay can be tailored to a specific set of models or support models that fit the size constraints without hardware reconfiguration.

2 LSTM BACKGROUND

Long Short Term Memory (LSTM) and **Gated Recurrent Units (GRUs)** are Recurrent Neural Networks (RNNs) that have been proposed to overcome the vanishing gradient problem in vanilla RNNs. Although both perform similarly in many tasks, the more complex structure of LSTMs theoretically allows them to learn more complex dependencies.

Equations (1) to (6) describe the operation of an LSTM unit, which is also illustrated in Figure 1, where $\odot$ denotes element-wise multiplication, and W and U are the weights of current input features and the previous cell outputs, respectively. More precisely, an LSTM unit consists of the forget gate (f_t), input gate (i_t), and output gate (o_t), along with the cell state (C_t), its partial result ($\tilde{C}_t$), and the LSTM cell output (H_t). The forget gate controls the amount of information to discard

from the previous cell state, the input and partial cell state define the new information to add to the cell state and the output gate defines the LSTM cell's output based on the current cell state.

An LSTM network can process a sequence of inputs, each of which can be a scalar or a vector. The sigmoid (σ) and tanh activation functions are most commonly used in this configuration. However, some flexibility in the application of activation functions is required to support different networks.

$$f_t = \sigma(W_f x_t + U_f H_{t-1} + b_f) \tag{1}$$

$$i_t = \sigma(W_i x_t + U_i H_{t-1} + b_i) \tag{2}$$

$$\tilde{C}_t = tanh(W_c x_t + U_c H_{t-1} + b_c) \tag{3}$$

$$o_t = \sigma(W_o x_t + U_o H_{t-1} + b_o) \tag{4}$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{5}$$

$$H_t = tanh(c_t) \odot o_t \tag{6}$$

The main difference between LSTMs and RNNs compared with other NNs, is the feedback connections from previous outputs (C_{t-1} and H_{t-1}). These dependencies restrict their performance while making routing, and dataflow in general, in custom architectures more complex. Nonetheless, common computing patterns in LSTMs and fully connected layers exist in Equations (1) to (4). A challenge in LSTMs however is the fact that the dimensions of W and U are not necessarily the same. The former depends on the number of input features and the number of units while the latter depends solely on the number of units of the LSTM. This translates to unbalanced latencies, when the two are computed separately. These two matrices can be concatenated into one, creating a single larger matrix with the same dimensions across all gates. This not only balances the compute latency within each gate but also makes the compute pattern of each gate the same as the the multiply accumulate operations in the fully connected layers. Each of Equations (1) to (4) can be mapped to a single neuron in a fully connected layer, thus, an LSTM unit occupies the equivalent of four neurons in a fully connected layer.

3 RELATED WORK

Previous related work that targets LSTMs in the embedded domain is presented in [4, 19–22], demonstrating the benefits of custom computing architectures in energy efficiency and performance, compared to software programmable computing platforms. Specifically, the work in [19] uses quantised (6–16 bit) LSTM models for speech recognition enabling their design to keep weights and intermediate results on chip and avoid energy consuming external memory accesses. The authors implement a matrix-vector multiplication unit that partially unrolls parallelism within an LSTM layer and is time-multiplexed for a complete layer computation. Their proposed implementation operates at 100 MHz on a Xilinx Zynq XC7Z045 FPGA and shows a distinct advantage in terms of energy efficiency compared to a high-end NVIDIA GeForce Titan X GPU.

The work in [20] presents a bi-directional LSTM for optical character recognition that uses 5-bit weights and fits in the on-chip memory of a Xilinx Zynq XC7Z045 device. The authors implement a single LSTM cell and unroll the computations of each gate in it, time multiplexing the instance according to the dimensions of each LSTM layer. In addition, the authors take advantage of the fact that in bi-directional LSTMs, two inputs are processed at a time and overlap their computations in order to alleviate the idle cycles between dependent LSTM iterations. The effectiveness of this approach is evaluated by implementing various designs, starting with a design that uses a single instance of their proposed architecture, exploring its scalability by instantiating six such computing blocks. For single input inference, the single instantiation design offers 152 GOPs

throughput at 166 MHz or 130 GOPs at 142 MHz. The design that incorporates six instances, operates at 142 MHz and obtains 308 GOPs for single input inference whereas for offline processing, in which batch processing with six images is used, 693 GOPs is achieved. The results show that the proposed approach does not scale well for single image inference, offering only a 2.4× scaling when instantiating six instances of the design. The proposed architecture scales better with batch processing, offering 5.3× the baseline throughput, however this is still not linear. The authors further expanded their work in [4] exploring extreme quantization methods using 1–8 bits. Their exploration yields a design that operates at 266 MHz,on a Xilinx Zynq UltraScale+ XCZU7EV, and offers throughput that ranges from 661 to 4201 GOPs for the different precisions used.

A unified LSTM accelerator flow is presented in [21], that takes as input a trained model and the target FPGA device specification and generates an accelerator design accordingly. The proposed flow generates an accelerator for each model and the accelerator is time multiplexed for each LSTM layer within a network. The generated accelerator does not support the activation function computations, which are offloaded to software on the ARM core. The LSTM gate results are therefore transferred to the off-chip memory, processed by the ARM core and then transferred back to the accelerator, resulting in frequent external memory transfers that are energy demanding and add a performance overhead. The overall SoC design uses the ARM core for coordinating the accelerator throughout LSTM computation, in addition to the activation function computation, thus not offering a self-contained accelerator solution. The authors implemented various versions of their accelerator, using 16 bit fixed point, 32 bit floating point and their equivalent pruned versions, on a Xilinx Zynq XC7Z020 FPGA operating at 150 MHz. The pruned equivalents reduced the inference time by 32% and 42% for the fixed and floating point implementations, respectively. Compared to the work in [23], the authors obtained about 10× improved inference time for the fixed point implementation whereas the floating point implementation offers negligible acceleration. Both implementations, however, demonstrate better power efficiency, being 11.7% and 0.32× more power efficient. The authors extend their evaluation by exploring the scalability of their floating point architecture by implementing a larger LSTM layer on a Xilinx Virtex VX485T, FPGA, obtaining 10.7 GFLOPs/s.

The mapping of large LSTM layers on Xilinx Virtex VX690T and Zynq 7Z045 FPGAs is explored in [22]. The authors aimed at optimising the matrix-vector multiplications and their dependencies in LSTMs with weight matrix partitioning and an optimised batch processing strategy. Their proposed approach is tailored for batch processing and uses 16-bit fixed point representation while operating at 125 MHz and 142 MHz on the Virtex 7 and Zynq devices, respectively. The authors obtained 356 GOPs on the Virtex 7 and 221 GOPs on the Zynq, demonstrating improved performance and energy efficiency compared to an Intel Xeon E5-2665 CPU, Nvidia TITAN X Pascal GPU, and other related previous work on FPGAs.

Other related work that targets the same LSTM models as in our evaluation, therefore better suited for direct comparisons, is described in [23–26]. The work in [23, 24] presents three different LSTM co-processors on an FPGA that balance memory bandwidth and internal storage utilization to optimize performance per unit power. The first streams all the necessary data from off-chip memory, the second stores all data on chip and the third is a more balanced design. The authors test their co-processors on a character level network comprising two LSTM layers, each with 128 units. The NN model used, includes a fully connected layer at the end that uses 65 neurons for the final classification, which has not been included in the architecture. All co-processors use Q8.8 fixed point representation and operate at 142 MHz on a Xilinx Zynq-7000 FPGA. The three architectures are compared in terms of resource utilization and memory bandwidth, and shown to provide orders of magnitude better performance per unit power compared to embedded processors,

with the design that stores all data on-chip being the most efficient in terms of performance per unit power.

A stochastic computing based LSTM implementation is presented in [25], focusing on reducing the hardware cost and power consumption of fundamental arithmetic components within an LSTM. The authors evaluate their approach on an LSTM layer with 16 hidden units, trained on the MNIST dataset. As with other previous work, the fully connected layer comprising 10 neurons was not included in the architecture. The authors implement their designs on a Xilinx Zynq-7000 FPGA, operating at 100 MHz, and make comparisons between the baseline and their two proposed designs in terms of power consumption, classification accuracy, and runtime. They show a trade-off between runtime and resource utilization and power, demonstrating their ability to scale to a suitable specification.

The authors in [26] propose a high throughput and energy efficient LSTM architecture utilizing an approximate multiplier. This results in a multiplierless implementation and effectively reduces power consumption and resource utilization, at the cost of multiple and variable clock cycles due to its data dependent nature. As a result, performance is less predictable. Hierarchical pipelining is used to improve performance by overlapping these computations. The proposed approach applies range-based linear quantization to a language model LSTM, with the same configurations as the model in [23, 24], on a Xilinx Zynq XC7Z030 FPGA. The implemented design uses 8-bit fixed point precision and operates at 100 MHz. The authors expanded their work in [27], in an ASIC implementation using 65 nm CMOS technology. The proposed ASIC implementation operates at 322 MHz while the low power approach resulted in an energy efficient implementation.

Related work can also be found in [10–13], in which the authors have focused on accelerating LSTM computation on more capable FPGAs with PCIe interconnect in servers. The approaches used, however, are not suitable for constrained edge devices, where fully unrolling computations cannot be achieved.

Additionally, algorithmic optimisations have been explored in [28, 29], where the authors take advantage of sparsity as a result of pruning to generate FPGA accelerators. The architecture in [29] achieves a frequency of 238 MHz, though results at 200 MHz were used for comparisons with previous work. The achieved 200 GOPs raw throughput translates to 22.2 GOPs/W raw energy efficiency. Taking into account the model sparsity, the effective throughput is 1600 GOPs, which in turn amounts to 177.9 GOPs/W energy efficiency. The work in [28] has explored pruning in more capable devices with PCIe interconnect, obtaining 200 MHz.

The majority of previous related work focuses solely on the LSTM computation, not including the implementation of fully connected layers, and thus do not provide a complete edge solution. Moreover, all reported operating frequencies are well below the devices' theoretical maximum, which results not only in lower performance, but also in lower energy efficiency due to leakage currents [10]. Meanwhile, generic NN accelerator architectures cannot implement LSTMs without modification, or suffer a significant performance and energy overhead due to the dependencies on previous outputs necessitating transfers to off-chip memory. This calls for more programmable custom computing architectures for LSTMs.

Architecture centric overlays were shown to offer high throughput since they utilize the underlying FPGA resources more efficiently [30]. The work in [31] described a streaming overlay for fully connected layers utilizing the DSP blocks of a Zynq Ultrascale+ ZU7EV FPGA. That design was shown to achieve close to the theoretical maximum frequency while using minimal resources, but supported only feed-forward networks with the ReLU activation function, and was not shown to scale. We propose a streaming overlay architecture that supports the computation of LSTM and Fully Connected layers, offering a complete edge solution, while supporting the more complex Sigmoid and Tanh activation functions through approximations. The overlay heavily exploits

programmable DSP block capabilities and is carefully designed to maintain short critical paths and relatively moderate routing complexity in order to achieve high operating frequency. At the same time, the overlay concept offers a more programmable solution, compared to fixed accelerators, allowing model parameters to be updated. The proposed architecture also operates in streaming mode, which is more responsive compared to batch processing and is therefore more suitable for devices at the edge.

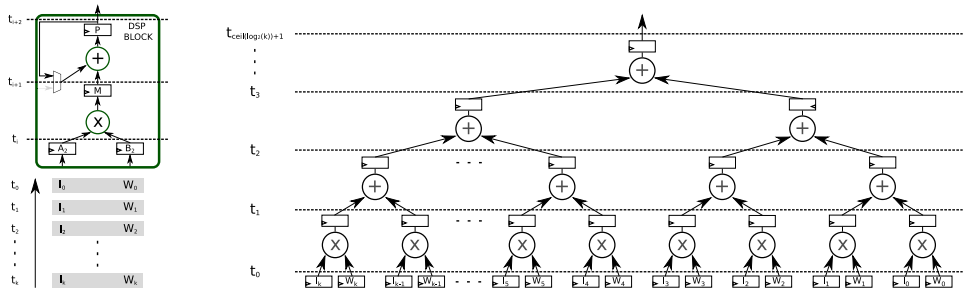
4 PROPOSED LSTM ARCHITECTURE

This section outlines the various design choices and operation of the main building blocks of the proposed streaming architecture that can be configured to implement LSTM or fully connected layers. The architecture uses DSP blocks for the neural network computations while also supporting other widely used settings in these layers (e.g., the option to return sequences in LSTM layers). Although systolic arrays are popular for inference architectures, as they are efficient for matrix-matrix multiplications, this works best for CNNs with batch processing rather than other NN structures including LSTMs. This batching helps reduce the overheads of loading weights. In streaming processing, as targeted by this work, systolic arrays would be less efficiently utilized due to the lack of batching and shared weights. Additionally pipeline parallelism would be harder to achieve due to the dependencies inherent in LSTMs. Hence, we adopt an alternative approach to implementing the multiply-accumulate operations, as outlined in the following sections.

4.1 Serial vs Fully Parallel Multiply Accumulate

While fully unrolling the individual multiplications followed by an adder tree, as shown in Figure 2(b), has been widely used to take advantage of parallelism in **multiply-accumulate (MAC)** operations in FIR filters and convolutions [32], it is not always ideal when considering streaming applications with a high degree of parallelism, for example in neural networks. FIR filters and convolutions are usually of smaller dimensions, compared to neural networks, having fewer coefficient storage requirements and workload to accelerate. The former does not hinder on-chip storage while the latter calls for more parallelism. MAC tree architectures are less flexible and adaptable to varying filter dimensions, being underutilized when computing a smaller filter or requiring complex partitioning of larger filters in order to fit. Meanwhile, latency for each pass remains the same. The computation at each layer of a neural network can be decomposed into a large vector-matrix multiplication that enables the sum of products for each neuron in the current layer with each in the previous layer and the corresponding weights. But this full unrolling is costly in terms of hardware, and the scale of these matrix multiplication units can hamper achievable frequency. Furthermore, this typically results in a layer-wise operation that necessitates significant transfers on/off chip between layers. These overheads are amortised by batch processing. Instead, we consider the fact that neural networks typically contain sufficient numbers of neurons to offer a coarser grained level of parallelism to exploit, where each neuron is mapped to a computational element, and its own results are calculated serially. This offers less dense signal connectivity and enables the compute units to operate at high frequency, while also affording flexibility to different network parameters, and avoiding memory transfers.

Our implemented serial computing architecture is ideally suited to LSTMs as it alleviates some of the inherent dependencies on previous outputs. Specifically, a subsequent LSTM iteration can be initiated as soon as 75% of the current iteration's outputs have been generated. This is because the implemented LSTM addon is able to generate one output every four clock cycles, meaning that the remaining 25% of the outputs will be generated by the time they are needed by the DSP blocks. LSTM computations using MAC trees on the other hand, require a previous iteration to be fully completed before starting the new one, since all the input data needs to be present in



(a) Serial Compute. (b) Fully Unrolled Multiply-Accumulate tree.

Fig. 2. Different compute architectures for the Multiply-Accumulate operation in Neural Networks.

Table 1. Latency and Resource Utilization of the Two Compute Methods

	k = 100		*k = 128*		*k = 256*	
	Latency (Clock Cycles)	DSP Blocks	Latency (Clock Cycles)	DSP Blocks	Latency (Clock Cycles)	DSP Blocks
Serial Compute	101	1 (100)	129	1 (128)	257	1 (256)
MAC tree	14	149	14	191	16	383

the registers for the new computation. The latter, coupled with the fact that it is very difficult to implement even a lightweight neural network on chip with fully parallel-fully unrolled MAC trees, without extreme optimisations, makes the MAC tree based architectures not ideal for edge processing, where we do not typically want batching to be applied to amortise overheads.

In addition, serial multiply-accumulate operation can be more efficiently mapped to a single DSP block, fully utilizing the multiplier and adder, while also requiring less memory for the intermediate results since they are consumed in a single register throughout the flow of inputs. This method is shown in Figure 2(a). In contrast, adders in direct MAC trees are usually implemented either using LUTs or DSP blocks. When implemented using LUTs, the adders consume FPGA resources that could otherwise be used to support the neuron operation (e.g., Control logic, memories in LUTRAMs, registers) while also potentially consuming more power, since functions implemented in a DSP block use less power compared their equivalent implementations in logic [33]. When implemented using DSP blocks, only the adder in each block is used, underutilizing the DSP block capabilities by leaving the multiplier unused, which may have a significant impact on efficiency for larger numbers of inputs. Systolic array implementations can efficiently utilize both components of the DSP block, but require considered data scheduling at the inputs to the array.

Table 1 shows the DSP utilization and latency for each of the two MAC architectures for a neuron with k inputs. For simplicity, we assume that the input data from the previous layer is already loaded in the registers while we also do not consider routing complexity, both of which favour the MAC tree architecture. Each multiplier along with each adder that follows in the first row of the MAC tree is considered to be mapped to a single DSP block, using their cascade interconnect [34]. To enable direct comparisons between the two compute methods, we assume that each of the adders that follow is mapped to a DSP block instead of FPGA fabric. This would enable the MAC tree to achieve higher operating frequency, as close as possible to the serial compute frequency, at the cost of underutilizing the DSP block capabilities by not using the multiplication while requiring two clock cycles latency for each addition. Starting with the MAC tree, its latency scales according to the next greatest power of two of k. The MAC tree offers latency improvements that range from 6.2× to 15.1× while consuming 148× to 382× more DSP blocks. We observe that

the benefits of the MAC tree in terms of latency are disproportional compared to the resources used, thus less efficient. Although it might be argued, under ideal circumstances, that the the MAC tree is able to generate a new output every clock cycle whereas the serial compute every k cycles, this can be compensated with the coarser, per neuron, parallelism. For example, for k inputs, exploiting parallelism for at least k neurons (numbers reported in brackets in Table 1) in a layer results in having the exact same throughput while consuming 27% to 33% fewer DSP blocks (assuming that there is sufficient parallelism within a layer to do so).

4.2 Proposed Neuron Architecture

Our architecture takes the neural network computation and does away with the matrix representation, instead opting for neuron-based parallelism where each neuron is implemented as a computational unit that processes its output in a serial manner. Each neuron is mapped to a single DSP block, supported by the required control logic and memory to enable it to fully implement the neuron's function. It operates in three modes: configuration, control, and compute. Initially, configuration takes place, in which all weights, biases and activation functions are set for each neuron. Once configuration is complete, compute and control operations run concurrently and input data starts to flow in. Outputs from the previous layer stream serially, one for each neuron from that layer at a time, and are multiplied in each neuron in the current layer by the corresponding weight stored in the weight memory. An address counter manages weight memory addressing and DSP block opmode selection. Each DSP block operates in one of two different opmodes, the first input-weight product is added to the configured bias, while subsequent products are accumulated with this sum. This is enabled by the dynamic DSP block control in modern Xilinx FPGAs [35]. This results in not having to reset the accumulation register before a new neuron computation and saves a clock cycle compared to adding the bias after the completion of multiply-accumulate. The proposed architecture is depicted in Figure 3, showing the details of a neuron compute unit among others in a layer. The serial dataflow of the overlay, coupled with the more minimal use of resources and fanouts, allows for a more scalable architecture in which each self-contained neuron can be replicated as many times as needed to form a layer, and each layer in turn to form a lightweight neural network on chip.

4.3 Neural Network Multiply-Accumulate

The *Neural Network Multiply-Accumulate architecture*, shown in Figure 4, consists of a series of DSP blocks, each calculating the multiply and accumulate operation. The inputs flow in each layer serially, multiplied by their corresponding weights and accumulated in each DSP block.

The word length of inputs, weights, and biases are defined to fit the DSP48E2 primitive in Xilinx UltraScale+ devices. Inputs are 27 bits, weights 18 bits, and biases are 16 bits. All word lengths are signed and use 11 fractional bits. The accumulation register within the DSP block is 48 bits and uses 22 fractional bits. Unrolling parallel operations at each neuron results in a naturally balanced workload between DSP blocks, while being more resource efficient by using both multipliers and adders within the DSP blocks. This unrolling scheme enables the overlay to map all the neurons of lightweight to moderate NNs on chip, and by accumulating all the intermediate results of each neuron within a single register, it reduces on chip memory requirements.

Moreover, this arrangement enables the serial flow of input data from neuron to neuron, which results in relatively low fanout, while also passing all input data to each neuron just once, avoiding additional storage and operational overhead to cache and re-flow previous input data. The use of DSP blocks coupled with the more compact dataflow result in a short critical path and more manageable routing which in turn enables a high frequency of operation. The *Neural Network*

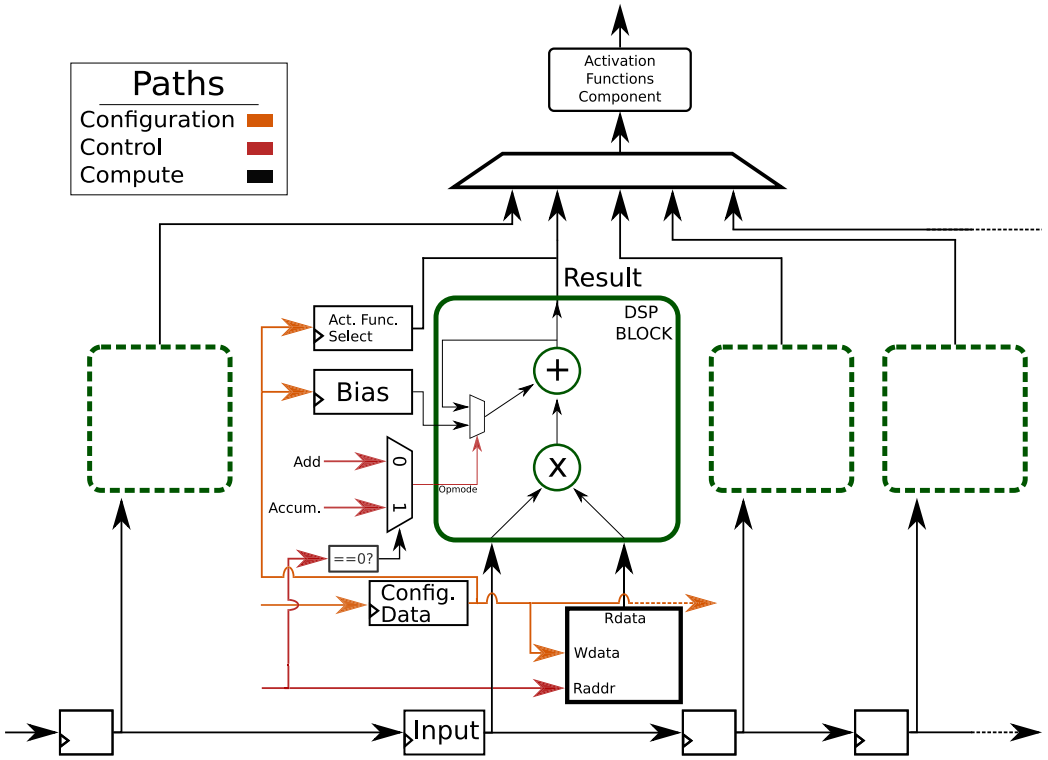


Fig. 3. Neuron architecture showing the configuration, control and compute paths.

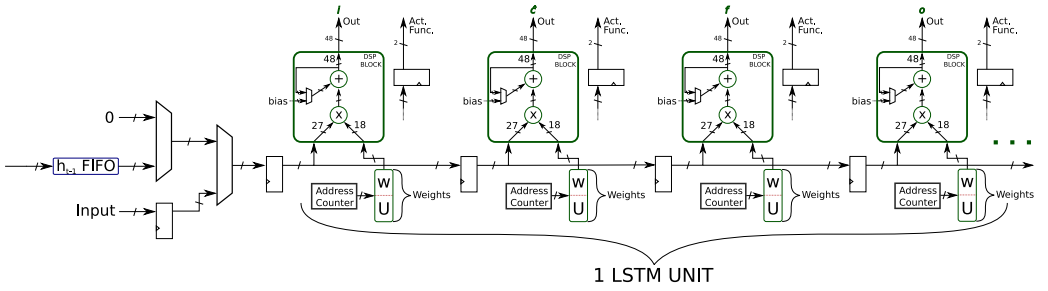


Fig. 4. Neural Network Multiply-Accumulate architecture.

MAC's operation deviates from the more mainstream acceleration methods on larger devices used for the matrix-vector operations (e.g., systolic arrays) to better suit the constraints of edge devices.

Due to the weight depth of each neuron within a layer being equal, neurons are expected to fire one after the other, propagating the serial dataflow to the following layers. In order for the serial flow to be maintained, only one neuron within a layer should fire at any given time. This means that if the number of inputs to a layer is less than the number of neurons in that layer, stall cycles are required to maintain this serial firing. The aforementioned data flow and stall operation are graphically depicted over time in Figure 5.

When the *Neural Network MAC architecture* is configured as a fully connected layer, the data flows serially to it from the input source. If it is configured as an LSTM layer, however, the new

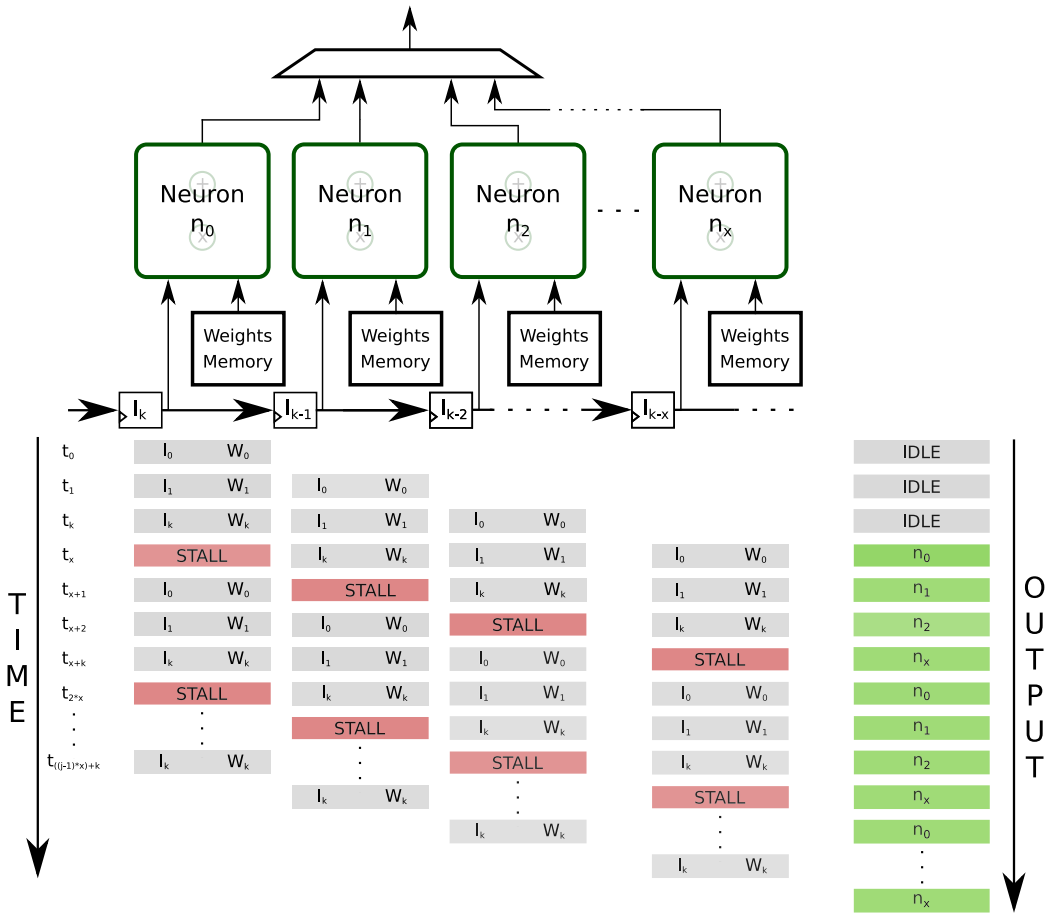


Fig. 5. Neural Network MAC serial flow and stalling operation.

input data flows in at first, while the previous output, H_{t-1}, stored in a FIFO, follows. Each of Equations (1) to (4) is mapped to a DSP block, with one LSTM unit occupying four DSP blocks. The input selection has been implemented with multiplexers and corresponding control logic. A 2-bit register sets the desired activation function for each neuron. The top level architecture supports the ReLU, approximated versions of Sigmoid and Tanh, and a passthrough datapath in case none is selected.

4.4 Activation Function Approximations

Various activations functions are used in neural networks for non-linearities, with ReLU, Sigmoid, and Tanh being the most widely used. Although activation functions may not be the most computationally complex part of neural network architectures, the use of exponents in Sigmoid and Tanh functions make them difficult to implement in embedded architectures.

Although ReLU is suitable for hardware implementation, various NN applications call for the use of Sigmoid or Tanh functions. For example, the forget gate in an LSTM layer uses the Sigmoid function, the output of which determines the percentage of information to be kept from the previous layer. This has led to the exploration of alternative ways to implement these functions more efficiently, especially in fault-tolerant, approximate computing applications. The majority

of previous neural network implementations map the activation functions in look-up-table memories, one for each function. With this approach, accuracy depends on the granularity of the look-up-table, with error being inversely proportional to the size of the table. This approach can use significant area in lightweight unrolled implementations, in which multiple tables are required. Furthermore, in architectures where multiple activation functions need to be supported, separate look-up-tables are required, of which only a subset are used at any one time. Other previous work has focused on piece-wise approximations of these functions [36], while others have approximated the active region of these functions linearly with cut-off regions [26]. Another example of the latter is the hard sigmoid activation function in Tensorflow [37]. More complex activation function architectures have also been presented in [38], where implementations in half and full precision floating point have been explored.

We propose activation function approximation using piecewise linear approximation while also considering how some coefficients can be modified to be more efficiently implemented in hardware. Moreover, since only one activation function is used at a time in each unit, common expressions are merged between the different activation functions in hardware. As a result, the logic required is minimized, contributing not only to reduced area but also improved performance. The proposed activation function architecture can be configured to any one of the most popular activation functions at runtime, without re-implementation or re-loading of a lookup table, while maintaining low area utilization and high performance.

```
def custom_sigmoid_hw(x):
  point_twenty_five = _constant_to_tensor(0.25, x.dtype.base_dtype)
  point_five = _constant_to_tensor(0.5, x.dtype.base_dtype)
  x = math_ops.mul(x, point_twenty_five)
  x = math_ops.add(x, point_five)
  x = clip_ops.clip_by_value(x, 0., 1.)
  return x

def custom_tanh_hw(x):
  point_seventy_five = _constant_to_tensor(0.75, x.dtype.base_dtype
      )
  x = math_ops.mul(x, point_seventy_five)
  x = clip_ops.clip_by_value(x, -1., 1.)
  return x
```

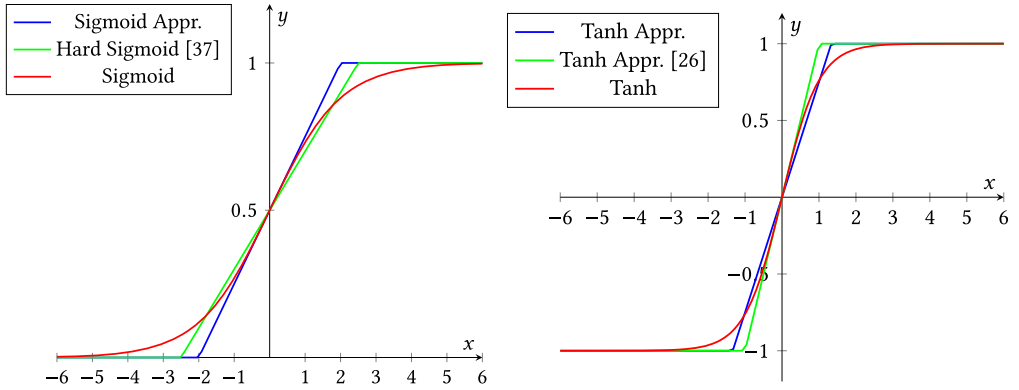
Listing 1. Approximation functions for Tensorflow.

4.4.1 Approximations Applied in Software. The most relevant previous work to ours in approximating the activation functions is summarized in Table 2, where *Sigmoid* and *Tanh* functions are bounded to 0,1 and −1,1, respectively. The approximated coefficients used in this work are modified slightly from those referenced to more efficiently suit the fixed point representation and hardware. Although the *Tanh* approximation used in [26] is simpler to implement, especially in hardware, it deviates more from the true function. All aforementioned approximations are presented graphically against the *Sigmoid* and *Tanh* in Figures 6(a) and 6(b).

The main difference in our work is that the approximations can be applied during training, in addition to post-training. Similarly to how the hard sigmoid is defined in Tensorflow, by changing the coefficients accordingly, the approximations can be defined as shown in Listing 1 and be used to train a model in floating point. The error introduced by the approximations is therefore taken into consideration during training and is alleviated, leaving more margin for error in fixed point representation.

Table 2. Approximated Functions Equations

Act. Func.	Equation	Work	MAE
Hard Sigmoid	y = 0.2x + 0.5	[37]	0.019
Approx. Sigmoid	y = 0.25x + 0.5	[26], This work	0.033
Approx. Tanh	y = x	[26]	0.088
Approx. Tanh	y = 0.75x	This work	0.063



(a) Sigmoid functions.

(b) Tanh functions.

Fig. 6. Activation functions and their approximations.

To demonstrate the effectiveness of this approach, we initially compare the approximations used in this work with those presented earlier in Table 2. We calculate the **Mean Absolute Error (MAE)** between the baseline functions and their approximations on 40 data points between −2 and 2 with a step of 0.1. As expected, the hard sigmoid approximation in Tensorflow generates less error compared to the approximation used in our design, however the *0.25x* has a far more straight forward fixed point representation and multiplication. Meanwhile, the *Tanh* approximation used in [26], although simpler to implement, generates more error compared to that used in this work. Moreover, since common computations exist between the two approximations, the additional complexity introduced in the *Tanh* approximation is mitigated.

We further explore the benefits of using the approximations during training by training a three layer LSTM for temperature forecasting, similar to that in [39]. We use the weather time series dataset from the Max Planck Institute for Biogeochemistry to train a network with two LSTM layers, with 64 and 32 units, respectively, and a fully connected layer for the output layer comprising one neuron. We used the RMSprop optimizer while measuring the loss using mean absolute error. The model is trained to receive the last 720 measurements that span over the last five days, and predict the temperature in 12 hours. Initially, a baseline model is trained for 10 epochs with Tensorflow v2.2 using the default activation functions while a variation of this model is trained using the approximated activation functions. We then run inference on the two models while also changing the activation functions of the baseline model to the approximations. The losses obtained from these three models are summarised in Table 3, showing that by using the approximated functions during training, loss is comparable to the original function implementations.

4.4.2 Activation Functions in Hardware. We present a hardware architecture in Figure 7 that supports the most widely used activation functions, ReLU, approximated Sigmoid, and Tanh, while

Table 3. Impact of Approximated Activation Functions on the Weather Forecast Dataset

Trained with	**Sigmoid-Tanh**		**Approx.**
Inf. using	Sig.-Tanh	Approx.	Approx.
Train Loss	0.3141	0.4358	0.3182
Validation Loss	0.3370	0.4283	0.3302
Test Loss	0.3575	0.5569	0.3751

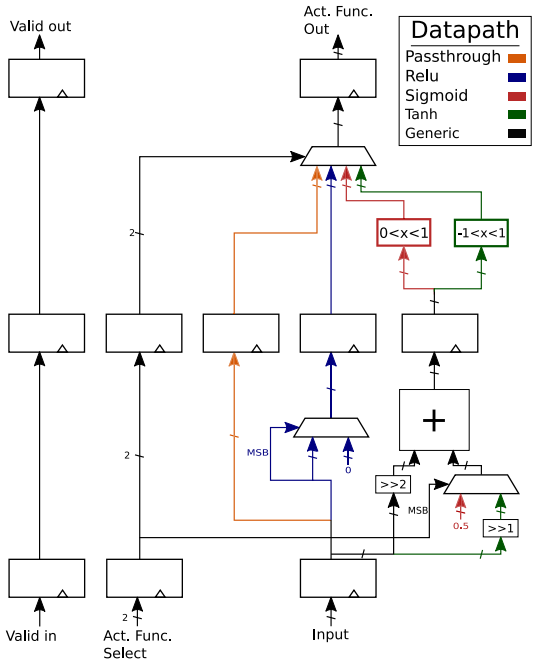


Fig. 7. Activation functions architecture.

also providing a passthrough path in case none is needed. A key feature is that common computations between Tanh and Sigmoid approximations are merged while all the multiplications are replaced by shifts and adds, avoiding the use of computationally expensive multiplications since the coefficients are fixed. Furthermore, since the output of the Sigmoid function is between 0 and +1, and the result of the Tanh function is between −1 and +1, the complexity of the component is not expected to grow significantly.

A parametrized architecture has been created in Verilog HDL, and implemented using Xilinx Vivado 2018.2 on a XCZU7EV Ultrascale+ device. We used various wordlengths and implemented these designs to make comparisons in terms of resource utilization. The results in Table 4 show that the proposed activation function architecture uses very few resources, less than 1% of the device, and these are able to operate at the device's maximum frequency.

4.5 LSTM Addon

The *LSTM Addon* computes Equations (5) and (6) of an LSTM layer. The results of i_t, $\tilde{C}_t$, f_t and o_t flow in serially, in this particular order. This data flow pattern repeats for each LSTM unit in the

Table 4. Resource Utilization of the Activation Functions Architecture

Bits	Fraction Bits	LUTs	Registers
16	8	36	86
27	12	74	141
32	16	90	166
48	24	132	246

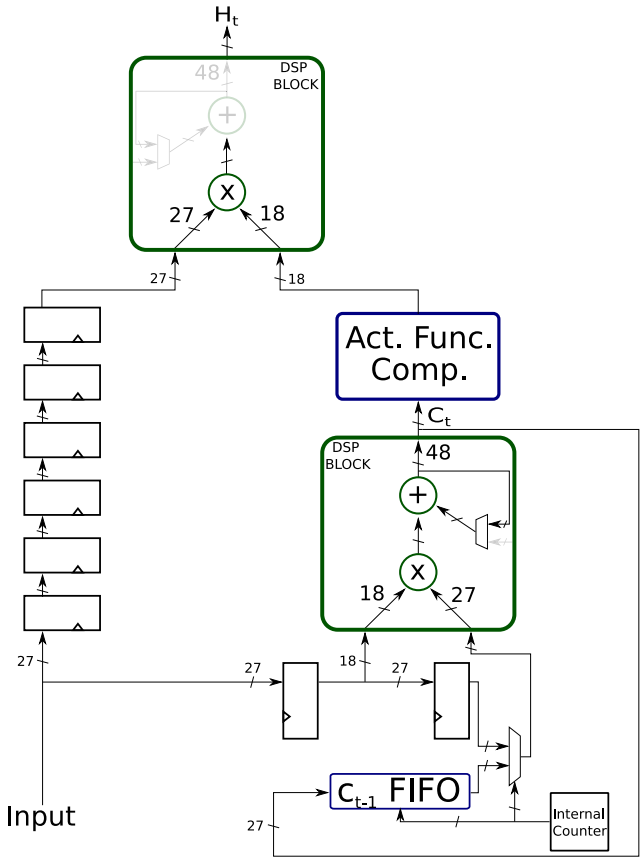


Fig. 8. LSTM Addon compute architecture.

Neural Network MAC. The i_t, $\tilde{C_t}$ and f_t are used by the datapath on the right in Figure 8, while o_t passes through delay registers and is used only by the DSP block at the output. Initially, i_t is multiplied by $\tilde{C_t}$, stored in the accumulation register of the DSP block, then the product of f_t and C_{t-1} that is read from the FIFO is calculated and added to accumulation register. This completes computation of Equation (5) and the result then fans-out to the C_{t-1} *FIFO*, where it is stored for the following timestep, and to the *Act. Func. Comp.*, where the activation function used in Equation (6) is applied. The DSP block at the output uses only the multiplier and completes the computation in Equation (6). An internal counter is used to synchronize all the operations and to reset the accumulation register of the DSP block between runs. Specifically, it increments with every valid

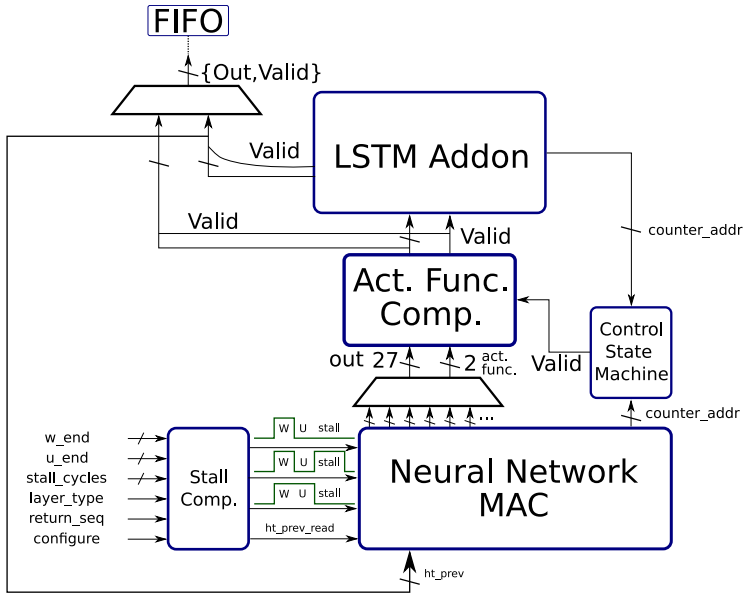


Fig. 9. Top level layer architecture.

input that passes through and selects which source is multiplexed into the DSP block. Initially, i_t is multiplied by $\tilde{C}_t$, where the counter selects the register to be passed as input to the DSP block. Then, f_t multiplied by C_{t-1} follows, where the counter selects the C_{t-1} FIFO as the source to the 27 bit input of the DSP block. The result of latter multiplication is accumulated to the DSP block to complete the computation of Equation (6).

4.6 Top Level Layer and Network Architecture

Figure 9 shows the top level layer architecture that includes all these functional blocks, along with control logic to synchronise and configure the dataflow for a single layer. A complete NN is formed by stacking multiple of these according to the network structure as shown in Figure 10, which shows the arrangement, interconnect, and interaction of the FIFOs with various building blocks. Each FIFO stores only the data required from the previous LSTM iteration, since the stored data is consumed by various compute blocks in the subsequent iteration, alleviating the need to store redundant data from more than one iteration as happens in many architectures that time multiplex their compute units. The largest network our architecture can fully support is roughly estimated in Equation (7). It is based on the number of layers, multiplied by the neurons in fully connected layers or number of units ×4 in LSTM layers, in addition to two DSP blocks used in each LSTM addon component.

$$\#DSP = (\#LSTM\ Units \times 4 + 2) \times \#Layers \tag{7}$$

This yields the number of required DSP blocks which should be fewer than what is available on the target device. Although our proposed approach ideally targets lightweight LSTM networks that can be fully unrolled at the neuron level, where its efficiency is maximised, it is also versatile enough to be implemented as a single layer-time multiplexed implementation or even folding parallel compute units, trading off performance and resource utilization to potentially adapt to larger networks.

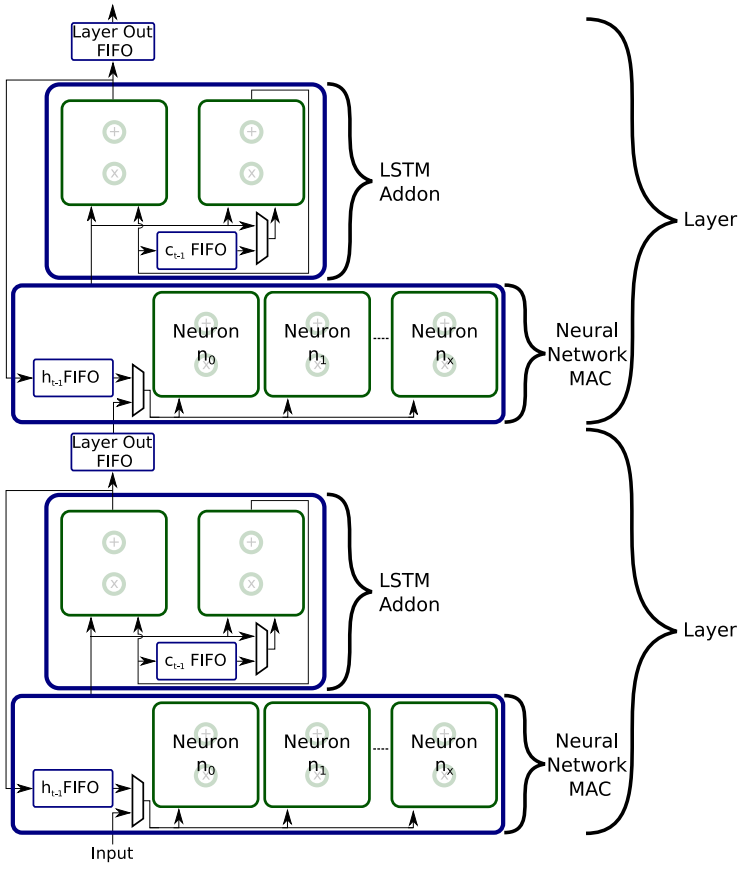


Fig. 10. Top level Neural Network architecture.

The main control blocks in the top level layer architecture are the *Stall Comp.* and the *Control State Machine*. The *Stall Comp.* is configured with the number of weights, stall cycles needed, the type of layers, the number of iterations and whether to return the sequences in LSTMs. It generates the control signals required in the *Neural Network MAC*, for example, to enable the address counters or which input source to choose from. Meanwhile, the *Control State Machine* synchronizes the flow between the *Neural Network MAC* and the other blocks.

A multiplexer after the *Neural Network MAC* selects which DSP block fires at each time step. Another at the output selects the appropriate datapath depending on whether it is an LSTM or fully connected layer. The *Neural Network MAC*'s serial operation, coupled with the ability to select the datapath according to the layer's configuration, through the use of multiplexers, forms an architecture that is able to adjust its latency and throughput for different neural network configurations. Although large scale multiplexers and decoders within the proposed architecture may cause delay, they can be pipelined according to the device's LUTs capabilities for high throughput at the cost of latency. Moreover, a FIFO is placed after each output multiplexer to gather the data required for the following layer.

The top level design that implements the whole NN is parametrised with the neural network configuration, i.e., number of layers, number of neurons in each layer, wordlengths, etc. This allows for a more flexible overlay that can also be ported to other devices that employ different DSP blocks.

Specifically, it supports any modern FPGA that incorporates the DSP48E2 or DSP48E1 primitives, which covers all Xilinx FPGAs currently available. DSP block instantiation, signal interconnection, and bit range is managed automatically using top level parameters that are passed to the various sub-modules. An overlay configuration can be used alongside a processor in an FPGA SoC. The processor can then configure the overlay at runtime with a specific network configuration. More importantly, weights and biases can also be set at runtime, enabling the overlay not only to adapt to weight updates and finetuning after deployment, but also to compute other LSTMs that fit within the bounds of the specified architecture. Underutilizing the overlay does however incur a performance overhead since data must still flow through unused layers.

4.6.1 Runtime Overlay Configuration. In addition to the weights and biases, the various control components of the overlay can be configured at runtime to accommodate different NN topologies that can be mapped to available overlay resources. Specifically, the programmable control mechanism in each layer is able to control the weight counter for each neuron, i.e., how many weights to read, and the number of neurons to iterate to within the layer. This effectively controls the number of weights and neurons in each layer. It also controls when to stall and for how many cycles, to support different layer dimensions. The weights and biases configuration in each neuron enables the overlay to accommodate different NNs or finetune an NN through weight updates. Although the number of layers cannot be changed at runtime, layers can be bypassed by multiplying the inputs with a weight of 1, adding 0 bias and selecting the pass-through activation function. The latter is effectively a trade-off between more complex routing at the cost of a few clock cycles overhead in terms of latency, which can be deemed negligible compared to the total clock cycles required for a complete network inference.

5 EVALUATION

5.1 Models for Evaluation

We use two LSTM models to evaluate our proposed architecture. These models have been used previously in [23–26], with which we make comparisons. While we are unable to use the exact code and framework, the models have been recreated to the best extent possible in Tensorflow v2.2 [37], according to the information provided in these references. The first model consists of a single LSTM layer with 16 units and a fully connected layer of 10 neurons. This network is trained as a classifier on the MNIST handwritten digit dataset, with the 10 output neurons corresponding to digits 0 to 9. The inputs to the LSTM model are 28×28 images with all pixel values normalised to a range from 0 to 1, this translates to 28 pixels being sent at a time for 28 iterations.

The second model is a character level LSTM trained on a part of Shakespeare's writing and comprises two LSTM layers, each with 128 units, and a fully connected layer with 65 neurons. The input to the LSTM is a vector with 65 one hot encoded values, each one representing a unique character that has been found in the text. A sequence of 50 of these vectors is passed to the LSTM model which generates the scores of the predicted 51st character at the output. The complexity of this model is representative of the real world application described in [15]. Specifically, the voice assistant NN described in [15], under our proposed compute scheme, would require about 960 DSP blocks, whereas the latter NN in our test case uses 1095 DSP blocks. We create an overlay architecture for each model for direct comparison with previous work, although the MNIST model could be run on the larger character overlay.

5.2 Impact of Approximated Activation Functions

We first evaluate how the use of the approximated activation functions impacts model accuracy for the models in our test case, as discussed in Section 4.4.

Table 5. Impact of the Approximated Activation Functions on the MNIST Dataset

Trained with	**Sigmoid-Tanh**		**Approx.**
Inf. using	Sig.-Tanh	Approx.	Approx.
Train (Loss/Accuracy)	0.0762/98.0%	0.984/80.7%	0.067/98.1%
Validation (Loss/Accuracy)	0.108/97.3%	0.944/81.7%	0.109/97.4%
Test (Loss/Accuracy)	0.106/97.5%	0.955/80.7%	0.117/97.4%

Table 6. Impact of the Approximated Activation Functions on the Shakespeare's Writing Dataset

Trained with	**Sigmoid-Tanh**		**Approx.**
Inf. using	Sig.-Tanh	Approx.	Approx.
Train Loss	0.8733	2.9	1.29

The results are presented in Tables 5 and 6. While we observe an increase in loss, along with a decrease in accuracy for the MNIST network, when the activation functions are simply switched after training, we then see that this can be alleviated by using the proposed activation functions during training. Although the loss in the character level LSTM can be considered high, we believe this is due to the learning complexity. More specifically, even though a low loss would mean that a model can work more accurately in inference, in this case it would mean that the model has memorised the textbook, which is a very difficult task. Instead, we would like the model to learn the coarser text patterns, rather than the finer details, and generate similar text.

5.3 Compute Overlap

The dependence of LSTM layers on previous outputs usually means the next iteration in a layer cannot start until the previous iteration has completed, reducing the parallel processing efficiency and effective throughput. The dataflow used in the proposed LSTM architecture coupled with unrolling parallelism at each neuron, enables the overlay to overlap part of the computations between iterations. In addition, the serial computing in the LSTM architecture enables the propagation of any compute configurations at the initial layer to the following layers, e.g., stall cycles applied in the first layer affect when the next layer will initiate its computation and so on.

The **Initiation Interval (II)** of the overlay is the clock cycle count between consecutive timestep computations in an LSTM layer. In other words, the number of clock cycles after which the next LSTM iteration computation can begin. Since each timestep in an LSTM layer depends on its previous outputs, the II essentially corresponds to the latency of the first layer in an unoptimised compute flow. The pipeline of the overlay is therefore underutilized between LSTM timestep executions in a naive execution, which in turn results in a less efficient and underperforming architecture for streaming applications. The latency of an LSTM layer, and therefore the unoptimised II, is modelled by Equation (8), in which the *#Inputs* and *#Previous Outputs* are initially passed through the first neuron. The first neuron requires 17 clock cycles to generate its output, followed by *#Units*4* clock cycles for the LSTM layer to generate all its outputs.

The serial neuron compute flow of the overlay enables the next timestep of an LSTM layer to begin with only part of its previous timestep outputs generated. Specifically, a new LSTM iteration starts with the new inputs flowing in the overlay serially, computing $W_y x_y$ in each of the Equations (1) to (4). The computation of the LSTM layer follows, which requires data from the previous timestep to compute $U_y H_{t-1}$ in each of the Equations (1) to (4). Since the proposed overlay compute flow generates an LSTM layer output every four clock cycles, the next iteration can start

Table 7. Overlapped Computing Compared to Unoptimised Execution

	Unoptimised		**Overlapped**	
	MNIST	Char. LSTM	MNIST	Char. LSTM
II-1st layer	125	722	80	530
Clock Cycles	3503	37131	2342	27723

Table 8. Weights to Input Size Ratio

	Number of coeff.		**Size in bits**	
	MNIST	Char. LSTM	MNIST	Char. LSTM
Weights (18 bits)	2976	238208	53568	4287744
Biases (16 bits)	74	1089	1184	17424
Inputs (27 bits)	784	3250	21168	87750
Coefficients to total data	–	–	**72.12%**	**98.00%**

as soon as 75% of the previous outputs have been generated. Therefore, in the overlapped computing scenario, the computation of the next timestep can start *#Inputs+#Units* earlier, with the floor of this equation being $\#Units \times 4$. The overlapped II is modelled in Equation (9) and requires an additional clock cycle in order for the last required output to be stored in the layer buffer.

$$LSTM\ Layer\ Latency = 17 + \#Inputs + \#Previous\ Outputs + (\#Units \times 4) \tag{8}$$

$$Overlapped\ II = 17 + \#Previous\ Outputs + (\#Units \times 3) + 1 \tag{9}$$

For the LSTM networks in our case study, the impact of overlapped computing on the II of the first layer is shown in the first row of Table 7, as the number of clock cycles required to initiate a new LSTM iteration. The second row of Table 7 shows how the II reduction impacts the latency of the complete network computation for a set of inputs. This is the clock cycle count for an end to end NN computation, including the generation of the final output results at the fully connected layer, of a whole image for the MNIST LSTM and 50 iterations of characters for the character level LSTM.

The ability of the proposed architecture to overlap part of the computation within consecutive LSTM layer iterations, leads to a latency reduction of 33% for the MNIST-LSTM and 25% for the character level LSTM.

5.4 Weight Stationary Architecture

Previous work in neural networks has explored a wide spectrum of optimisations in an effort to reduce off-chip memory bandwidth, from computing parts of the neural network in batches and transferring weights accordingly, to extreme quantization of weights. In this architecture, the compute units, that typically consume most FPGA resources, are mapped to DSP blocks. This, coupled with the minimalistic approach in the design of other building blocks, results in releasing FPGA resources that can be used to store more weights on chip, maximising the device's storage capabilities. In Table 8, we show the total weight and data sizes for a single classification for the two networks. The weights and biases in the MNIST LSTM amount to about 73% of total data and about 98% for the character level LSTM. Architectures that store these parameters in off-chip memory require high bandwidth to achieve high throughput, while our approach reduces bandwidth requirements and potentially energy consumption.

Table 9. Resource Utilization and Performance Comparisons with Same Models

	MNIST LSTM				Character LSTM			
	Baseline [25]	Appr. A [25]	Appr. B [25]	**This work**	[24]	Deepstore [23]	Appr. [26]	**This work**
FPGA	XC7Z020	XC7Z020	XC7Z020	**XCZU7EV**	XC7Z020	XC7Z045	XC7Z030	**XCZU7EV**
Precision	N/A	N/A	N/A	**16-27 fixed**	16 fixed	16 fixed	8 fixed	**16-27** fixed
LUTs	7741 (14.55%)	9529 (17.91%)	6763 (12.71%)	**4244(1.84%)**	7201 (13.54%)	N/A	23036 (29.31%)	**95263 (41.35s%)**
Flip-Flops	2412 (2.27%)	8456 (7.95%)	5928 (5.57%)	**9308(1.75%)**	12960 (12.18%)	N/A	28481 (18.12%)	**118261 (25.66%)**
BRAM	3.58KB	0	0	**0 (0%)**	16 (11.43%)	N/A	180 × 36KB (67.92%)	**518 × 18KB(83.01%)**
DSP	1 (0.45%)	0 (0%)	0 (0%)	**78(4.51%)**	50 (22.73%)	N/A	0 (0%)	**1095(63.37%)**
Freq. (MHz)	N/A	100	100	**640**	142	142	100	**420**
DSP Max (MHz)	464	464	464	**775**	464	650	548-742	**775**
DSP Freq. (%)	N/A	21.6	21.6	**82.6**	30.6	21.8	18.2-13.5	**54.1**
Power (w)	0.142[e]	0.072[e]	0.038[e]	**0.679[e]**	1.94	2.3	1.19[e]	**8.531[e]**
Latency (ms)	0.17[a,c]	18.58[a,c]	18.58[a,c]	**0.0037[d]**	0.932[a,c]	N/A	N/A	**0.066[d]**
T'put-LSTM (GOPs)	0.928	0.00847	0.00847	**44.5**	0.29	1.05	8.08(max)/2.26(avg)	**363.7**
T'put (class./s)	5882.4[b,c]	53.8[a,b]	53.8[a,b]	**282186.9[d]**	1073.0[a,b]	1969.0 [a,b]	N/A	**15819.2[d]**
T'put (GOPs)	N/A	N/A	N/A	**44.6**	N/A	N/A	N/A	**363.9**
Efficiency (GOPs/W)	6.54[a]	0.12[a]	0.22[a]	**65.67[d]**	0.15[a]	0.46[a]	6.79(max)/1.89(avg)[a]	**42.7[d]**

[a] LSTM layers only.
[b] Estimated from the reported average runtime.
[c] Reported as average runtime.
[d] Complete network (i.e., including FC layers).
[e] Vivado power estimator.

Table 10. Resource Utilization and Performance Comparisons with Different Models

	Single Instance [20] (Streaming)	6 Instances [20] (Streaming)	6 Instances [20] (Batch)	[4]	[22]	[22]	[21]
FPGA	XC7Z045	XC7Z045	XC7Z045	XCZU7EV	VX690T	XC7Z045	XC7Z020
Precision	5 fixed	5 fixed	5 fixed	1-8 fixed	16 fixed	16 fixed	16 fixed
LUTs	32815 (15%)	161574 (74%)	190036 (87%)	N/A	204000 (47.0%)	166000 (75.8%)	51604 (97%)
Flip-Flops	14532 (3%)	51213 (12%)	78516 (18%)	N/A	222000 (25.6%)	150000 (34.4%)	69160 (65%)
BRAM	83 (15%)	339 (62%)	498 (91%)	N/A	1070 (72.8%)	517.5 (94.9%)	179.2 (64%)
DSP	33 (4%)	195 (22%)	198 (22%)	N/A	2060 (57.0%)	900 (100%)	180.4 (82%)
Freq. (MHz)	166	142	142	266	125	142	150
DSP Max (MHz)	548-742	548-742	548-742	775	548-741	650	464
DSP Freq. (%)	30.3-22.4	25.9-19.3	25.9-19.3	34.3	22.8-16.9	21.8	32.3
T'put (GOPs)	152.16[b]	308.05[b]	693.12[b]	746-4201[b]	356	221	4.25
Power (W)	1.7 [a,b]	6.97[a,b]	12.46[a,b]	N/A	26.5	10.6	2.29
Efficiency (GOPs/W)	89.52[a,b]	44.22[a,b]	55.62[a,b]	N/A	13.48	20.84	1.86

[a] Calculated from authors' reported results.
[b] Complete network (i.e., including FC layers).

5.5 Performance, Resource Utilization, and Comparisons

We implement two versions of our proposed overlay architecture, one for each NN, on a ZU7EV FPGA as found on the Xilinx Zynq Ultrascale+ ZCU104 board. Both overlays have been implemented in Verilog HDL using Xilinx Vivado 2018.2. Moreover, both have been integrated in an SoC implementation with the Arm Cortex A53 on the device and functionally verified, baremetal, on part of the datasets using Xilinx SDK. To enable the integration of the high operating frequency overlay in the SoC and overcome the lower operating frequency of required IPs, our overlay employs a dual clock configuration. A high frequency clock is used for the overlay's compute mode, whereas a slower clock that operates at a quarter of the fast clock frequency is used to configure the overlay and to transfer the input and output data. To match the rate of the slow clock input data with the fast clock compute, four inputs are transferred at a time at the slow clock rate to a dual clock FIFO, subsequently each input slot is extracted at the fast clock rate.

All results presented in this section are post-place and route for the overlay module only, extracted from the hierarchical results of the implemented SoC. Tables 9 and 10 compare the attributes of our proposed approach to previous work. To enable more objective comparisons with work targeting different FPGA devices, we derive attributes for overall efficiency. In addition, we report the theoretical maximum frequency of the DSP blocks for those devices, as found in the

devices' datasheets [40–43], and the frequencies achieved. In cases where the device's speed grade is not reported by the authors, we use a range of highest and lowest speed grades. Although the MNIST model can be computed within the larger overlay, we provide an overlay for each LSTM model to enable objective comparisons with previous work and to have a benchmark on how our overlay scales. The MNIST overlay would consume 6.8% of the DSP blocks and about 1.24% of the implemented memories of the character level LSTM overlay.

In addition to the MNIST overlay reported in Table 9, which uses LUTRAMs for the neuron memories, we have explored the use of BRAMs in an identical architecture. Naturally, this resulted in varying utilization of memory elements on the FPGA, but importantly, this also resulted in a reduced frequency and higher power estimation for the BRAM based overlay. The BRAM based overlay is able to compute at 520 MHz and configured at 130 MHz, whereas the LUTRAM based overlay is able to compute at 640 MHz and configured at 160 MHz. Meanwhile, the power estimation for the BRAM is higher, amounting to 0.845W compared to 0.679W the LUTRAM based overlay, which in turn results in poorer efficiency.

This suggests that the weight memories in lightweight and shallow neural networks are more performance and energy efficient when mapped to LUTRAMs, which can be partly due to the fact that only a small percentage of each BRAM bank is utilized. Meanwhile the simple routing of small FPGA fabric based memory offers better operating frequency. The configuration of the weights and biases takes place in streaming mode, using the slower clock, and is estimated to be around 0.02ms with negligible difference between the two overlays.

Our proposed overlay for the character level LSTM model is implemented using a hybrid memory arrangement in the first two LSTM layers, e.g., one neuron uses LUTRAM memory, the other BRAM memory, etc, while the output layer uses LUTRAM memories. A uniform memory overlay with either type of resource does not fit in the device when integrated with the SoC. Similarly to the MNIST overlay, the weights and biases are configured in streaming mode, using the slower clock, and this is estimated to take 2.29ms. Considering the BRAM based MNIST overlay as the baseline, the character level overlay scales well as the frequency is reduced by 19.2% while utilizing 14× to 37× more resources.

Table 9 summarises other relevant previous work that implements the exact same LSTM models as in our work. Our proposed MNIST overlay is competitive in terms of resource utilization with the work in [25] which focuses on approximate computing to yield multiplierless-low power implementations, while significantly outperforming in terms of latency, throughput, and efficiency. In addition, our proposed architecture achieves 82.6% the theoretical DSP block maximum frequency, compared to 21.6% achieved in [25]. Regarding the more complex character level LSTM overlay, although it utilizes more resources due to the higher neuron and layer parallelism and higher precision, our implementation is significantly better in terms of latency, throughput, and efficiency. Specifically, it is 22.6× more efficient compared to the average performance of the most competitive previous work in [26]. Meanwhile, our proposed architecture operates at 54.1% of the DSP block theoretical maximum frequency in the target FPGA, compared to 30.6% of the most competitive previous work in [24].

Although achieving high frequency does not necessarily result in the best overall performance, it is a good proxy for how well designed and efficient the architecture is on the underlying FPGA. In addition, we have shown how this operating frequency along with the various architectural and algorithmic optimisations, translate to overall performance metrics (e.g., latency and throughput).

Finally, we supplement our evaluation by porting the overlay to Xilinx Zynq 7000 devices, as reported in Table 11, for direct comparisons with previous work. These results demonstrate that our approach is effective on older devices as well. Specifically, we see that our MNIST overlay achieves 208 MHz, which is 44.83% of the devices' theoretical maximum. Compared to previous work using

Table 11. Resource Utilization and Frequency on Zynq 7000 Series

	MNIST LSTM	Character LSTM
FPGA	XC7Z020	XC7Z100
Precision	16-27 fixed	16-27 fixed
LUTs	4120 (7.74%)	93920 (33.86%)
Flip-Flops	8972 (8.43%)	113838 (20.52%)
BRAM	0 (0%)	259 (34.30%)
DSP	78 (35.45%)	1095 (54.21%)
Freq. (MHz)	208	312
DSP Max (MHz)	464	650
DSP Freq.%	44.83%	48%

MNIST in Table 9, the proposed overlay operates twice as fast on the same device. Furthermore, the character level overlay achieves 312 MHz, which is 48% of the theoretical maximum (650 MHz) of the XC7Z100 device and is approximately 2.2–3× faster compared to relevant previous work in Table 9.

We extend our comparisons with previous work in the embedded domain in Table 10. Although these implementations do not target the same models, they use various architectural approaches of interest on more modern FPGA devices. The implementations in [4, 20] target a Bidirectional-LSTM model for optical character recognition. The complete network consists of a single Bidirectional LSTM layer with a total of 200 nodes followed by a fully connected layer, both of which have been implemented in the compute architecture. Compared to our character level LSTM, this model is less complex in terms of LSTM cells used, number of layers, and precision, using 68.8–81.5% reduced precision. Nonetheless, our overlay architecture obtains better throughout compared to the single instance and six instances that operate in streaming mode. Although the single instance in [20] is more efficient, its performance does not scale well when six instances are implemented on the device, resulting in reduced efficiency which is slightly better than our proposed overlay. Its corresponding batch processing implementation with six instances yields improved throughput, compared to our streaming operation, while increasing its efficiency. This shows that streaming processing is more challenging to optimise, since it doesn't scale linearly with the increase of compute resources, whereas batch processing scales better with the availability of more input data. The authors further expanded their work in [4] for a more systematic exploration of the trade-offs of reduced precision, improving their obtained throughput significantly. The work in [22] aims at partitioning large LSTM layers and achieves the obtained throughput efficiency with a batch size of 64, which our proposed overlay outperforms. Lastly, the authors in [21] target a single LSTM layer only, with a similar configuration to the first layer of the character level LSTM, while some of the computations are offloaded to the ARM core, obtaining lower performance and efficiency with, however, a less capable device. Regarding the operating frequencies of previous work in Table 9, the most competitive one operates at 32.3% of the device's DSP block theoretical maximum, while our least competitive overlay operates at 54.1%. This demonstrates the frequency gains of our proposed approach, irrespective of the different FPGAs used in previous work.

6 CONCLUSIONS AND FUTURE WORK

We present a streaming overlay architecture based on DSP blocks that is able to compute lightweight to moderate sized LSTM and fully connected layers, storing all required weights on chip. Our approach is aimed at enhancing programmability and flexibility by using the overlay

concept, while providing high performance and resource efficiency by employing an architecture that achieves high operating frequency and consumes data serially. Its serial data flow, parallel neuron computation, and pipelined operation coupled with optimizations in compute overlap and on chip weight storage result in high throughput operation. The low level operation of the architecture is abstracted to form an overlay that can be configured at the top level with model configurations. Moreover, the implemented overlay, can be configured after deployment, at runtime, with the different model configurations, weights, and biases. Our experiments have shown that the standalone overlay architecture can operate at much higher frequencies in an SoC design, alongside the ARM core, within which it has been implemented and functionally verified. We show how our overlay architecture is competitive in terms of efficiency and outperforms other generic previous work in stream processing, while using higher precision. In the future, we intend to investigate reduced precision, along with pruned models, and explore how they can be efficiently mapped to the proposed overlay. Furthermore, we plan on expanding our approach to a framework with software support for an end to end NN overlay generation, able to partition and schedule the execution of large networks that may not fit on a device.

REFERENCES

[1] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2017. Show and tell: Lessons learned from the 2015 MSCOCO image captioning challenge. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39, 4 (2017), 652–663. DOI: http://dx.doi.org/10.1109/TPAMI.2016.2587640

[2] Lenos Ioannou and Suhaib A. Fahmy. 2019. Network intrusion detection using neural networks on FPGA SoCs. In *International Conference on Field Programmable Logic and Applications (FPL)*. 232–238. DOI: http://dx.doi.org/10.1109/FPL.2019.00043

[3] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Advances in Neural Information Processing Systems*, Vol. 29. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2016/file/d8330f857a17c53d217014ee776bfd50-Paper.pdf.

[4] Vladimir Rybalkin, Alessandro Pappalardo, Muhammad Mohsin Ghaffar, Giulio Gambardella, Norbert Wehn, and Michaela Blott. 2018. FINN-L: Library extensions and design trade-off analysis for variable precision LSTM networks on FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)*. 89–96. DOI: http://dx.doi.org/10.1109/FPL.2018.00024

[5] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both weights and connections for efficient neural networks. In *International Conference on Neural Information Processing Systems (NIPS)*. 1135–1143.

[6] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. 2017. Trained ternary quantization. In *International Conference on Learning Representations (ICLR)*.

[7] Allan Skillman and Tomas Edsö. 2020. A technical overview of Cortex-M55 and Ethos-U55: Arm's most capable processors for endpoint AI. In *IEEE Hot Chips Symposium*. DOI: http://dx.doi.org/10.1109/HCS49909.2020.9220415

[8] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2019. Survey and benchmarking of machine learning accelerators. In *IEEE High Performance Extreme Computing Conference (HPEC)*. DOI: http://dx.doi.org/10.1109/HPEC.2019.8916327

[9] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. 2018. Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions. *ACM Comput. Surv.* 51, 3 (2018), 56:1–56:39.

[10] Ephrem Wu, Xiaoqian Zhang, David Berman, and Inkeun Cho. 2017. A high-throughput reconfigurable processing array for neural networks. In *International Conference on Field Programmable Logic and Applications (FPL)*. DOI: http://dx.doi.org/10.23919/FPL.2017.8056794

[11] Ananda Samajdar, Tushar Garg, Tushar Krishna, and Nachiket Kapre. 2019. Scaling the cascades: Interconnect-aware FPGA implementation of machine learning problems. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 342–349. DOI: http://dx.doi.org/10.1109/FPL.2019.00061

[12] Eriko Nurvitadhi, Dongup Kwon, Ali Jafari, Andrew Boutros, Jaewoong Sim, Phillip Tomson, Huseyin Sumbul, Gregory Chen, Phil Knag, Raghavan Kumar, Ram Krishnamurthy, Sergey Gribok, Bogdan Pasca, Martin Langhammer, Debbie Marr, and Aravind Dasu. 2019. Why compete when you can work together: FPGA-ASIC integration for persistent RNNs. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 199–207.

[13] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa

Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A configurable cloud-scale DNN processor for real-time AI. In *International Symposium on Computer Architecture (ISCA)*. DOI: http://dx.doi.org/10.1109/ISCA.2018.00012

[14] Chen, Jiasi and Ran, Xukan. 2019. Deep learning with edge computing: A review. *Proc. IEEE* 107, 8 (2019), 1655–1674. DOI: http://dx.doi.org/10.1109/JPROC.2019.2921977

[15] Hey Siri: An On-device DNN-powered Voice Trigger for Apple's Personal Assistant. ([n. d.]).

[16] Martin Milenkoski, Kire Trivodaliev, Slobodan Kalajdziski, Mile Jovanov, and Biljana Risteska Stojkoska. 2018. Real time human activity recognition on smartphones using LSTM networks. In *International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 1126–1131. DOI: http://dx.doi.org/10.23919/MIPRO.2018.8400205

[17] Xiangyun Qing and Yugang Niu. 2018. Hourly day-ahead solar irradiance prediction using weather forecasts by LSTM. *Energy* 148 (2018), 461–468. DOI: http://dx.doi.org/10.1016/j.energy.2018.01.177

[18] Jihyun Kim, Jaehyun Kim, Huong Le Thi Thu, and Howon Kim. 2016. Long short term memory recurrent neural network classifier for intrusion detection. In *International Conference on Platform Technology and Service (PlatCon)*. DOI: http://dx.doi.org/10.1109/PlatCon.2016.7456805

[19] Minjae Lee, Kyuyeon Hwang, Jinhwan Park, Sungwook Choi, Sungho Shin, and Wonyong Sung. 2016. FPGA-based low-power speech recognition with recurrent neural networks. In *IEEE International Workshop on Signal Processing Systems (SiPS)*. 230–235. DOI: http://dx.doi.org/10.1109/SiPS.2016.48

[20] Vladimir Rybalkin, Norbert Wehn, Mohammad Reza Yousefi, and Didier Stricker. 2017. Hardware architecture of bidirectional long short-term memory neural network for optical character recognition. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 1390–1395. DOI: http://dx.doi.org/10.23919/DATE.2017.7927210

[21] Xinyi Zhang, Weiwen Jiang, and Jingtong Hu. 2020. Achieving full parallelism in LSTM via a unified accelerator design. In *IEEE International Conference on Computer Design (ICCD)*. 469–477. DOI: http://dx.doi.org/10.1109/ICCD50377.2020.00086

[22] Zhiqiang Que, Yongxin Zhu, Hongxiang Fan, Jiuxi Meng, Xinyu Niu, and Wayne Luk. 2020. Mapping large LSTMs to FPGAs with weight reuse. *J. Signal Process. Syst.* 92, 9 (2020), 965–979. DOI: http://dx.doi.org/10.1007/s11265-020-01549-8

[23] Andre Xian Ming Chang and Eugenio Culurciello. 2017. Hardware accelerators for recurrent neural networks on FPGA. In *IEEE International Symposium on Circuits and Systems (ISCAS)*. DOI: http://dx.doi.org/10.1109/ISCAS.2017.8050816

[24] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. 2015. Recurrent neural networks hardware implementation on FPGA. *CoRR* abs/1511.05552 (2015). arXiv:1511.05552. http://arxiv.org/abs/1511.05552.

[25] Guy Maor, Xiaoming Zeng, Zhendong Wang, and Yang Hu. 2019. An FPGA implementation of stochastic computing-based LSTM. In *IEEE International Conference on Computer Design (ICCD)*. 38–46. DOI: http://dx.doi.org/10.1109/ICCD46524.2019.00014

[26] Elham Azari and Sarma Vrudhula. 2019. An energy-efficient reconfigurable LSTM accelerator for natural language processing. In *IEEE International Conference on Big Data (Big Data)*. 4450–4459. DOI: http://dx.doi.org/10.1109/BigData47090.2019.9006030

[27] Elham Azari and Sarma B. K. Vrudhula. 2020. ELSA: A throughput-optimized design of an LSTM accelerator for energy-constrained devices. *ACM Trans. Embed. Comput. Syst.* 19, 1 (2020), 3:1–3:21. DOI: http://dx.doi.org/10.1145/3366634

[28] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. 2018. C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 11–20.

[29] Seyed Abolfazl Ghasemzadeh, Erfan Bank Tavakoli, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. 2021. BRDS: An FPGA-based LSTM accelerator with row-balanced dual-ratio sparsification. *CoRR* abs/2101.02667 (2021). arXiv:2101.02667. https://arxiv.org/abs/2101.02667.

[30] Abhishek K. Jain, Douglas L. Maskell, and Suhaib A. Fahmy. 2016. Throughput oriented FPGA overlays using DSP blocks. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 1628–1633.

[31] Lenos Ioannou and Suhaib A. Fahmy. 2019. Lightweight programmable DSP block overlay for streaming neural network acceleration. In *International Conference on Field-Programmable Technology (FPT)*. 355–358. DOI: http://dx.doi.org/10.1109/ICFPT47387.2019.00066

[32] Lenos Ioannou, Abdullah Al-Dujaili, and Suhaib A. Fahmy. 2020. High throughput spatial convolution filters on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 6 (2020), 1392–1402. DOI: http://dx.doi.org/10.1109/TVLSI.2020.2987202

[33] Xilinx. 2020. *UG579 UltraScale Architecture DSP Slice-User Guide.*

[34] 2018. *7 Series DSP48E1 Slice User Guide (UG479).*

[35] Bajaj Ronak and Suhaib A. Fahmy. 2016. Mapping for maximum performance on FPGA DSP blocks. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* 35, 4 (2016), 573–585.
[36] Xie Zhen-zhen and Zhang Su-yu. 2012. A non-linear approximation of the sigmoid function based FPGA. In *International Conference on Informatics, Cybernetics, and Computer Engineering (ICCE)*. 125–132.
[37] Martín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/.
[38] B. Pasca and M. Langhammer. 2018. Activation function architectures for FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)*. 43–50. DOI: http://dx.doi.org/10.1109/FPL.2018.00015
[39] Francois Chollet. 2017. *Deep Learning with Python*. Manning.
[40] 2020. *Zynq-7000 SoC (Z-7007S, Z-7012S, Z-7014S, Z-7010, Z-7015, and Z-7020) (DS187)*.
[41] 2018. *Zynq-7000 SoC (Z-7030, Z-7035, Z-7045, and Z-7100): DC and AC Switching Characteristics (DS191)*.
[42] 2021. *Virtex-7 T and XT FPGAs Data Sheet: DC and AC Switching Characteristics (DS183)*.
[43] 2021. *Zynq Ultrascale+ MPSoC Data Sheet: DC and AC Switching Characteristics (DS925)*.

Received 31 October 2021; revised 10 March 2022; accepted 26 May 2022