Spartus: A 9.4 TOp/s FPGA-Based LSTM Accelerator Exploiting Spatio-Temporal Sparsity

Chang Gao, Member, IEEE, Tobi Delbruck, Fellow, IEEE, and Shih-Chii Liu, Fellow, IEEE

Abstract—Long short-term memory (LSTM) recurrent networks are frequently used for tasks involving time-sequential data, such as speech recognition. Unlike previous LSTM accelerators that either exploit spatial weight sparsity or temporal activation sparsity, this article proposes a new accelerator called “Spartus” that exploits spatio-temporal sparsity to achieve ultralow latency inference. Spatial sparsity is induced using a new column-balanced targeted dropout (CBTD) structured pruning method, producing structured sparse weight matrices for a balanced workload. The pruned networks running on Spartus hardware achieve weight sparsity levels of up to 96% and 94% with negligible accuracy loss on the TIMIT and the Librispeech datasets. To induce temporal sparsity in LSTM, we extend the previous DeltaGRU method to the DeltaLSTM method. Combining spatio-temporal sparsity with CBTD and DeltaLSTM saves on weight memory access and associated arithmetic operations. The Spartus architecture is scalable and supports real-time online speech recognition when implemented on small and large FPGAs. Spartus per-sample latency for a single DeltaLSTM layer of 1024 neurons averages 1 μs. Exploiting spatio-temporal sparsity on our test LSTM network using the TIMIT dataset leads to 46× speedup of Spartus over its theoretical hardware performance to achieve 9.4 TOp/s effective batch-1 throughput and 1.1 TOp/sW power efficiency.

Index Terms—Delta network, dropout, edge computing, recurrent neural network (RNN), spiking neural network, structured pruning.

I. INTRODUCTION

R ECURRENT neural networks (RNNs) are widely used in tasks that involve temporal sequences. RNN variants, such as the long short-term memory (LSTM) [1] and gated recurrent unit (GRU) [2] models, use additional gating units to mitigate the problem of vanishing gradient. These variants achieve state-of-the-art prediction accuracy in tasks involving input temporal sequences, such as automatic speech recognition [3], [4] and natural language processing [5]. RNNs are also useful in latency-critical real-time control tasks, such as robotic prostheses control [6], gaming AI [7], and autonomous driving [8], which require the hardware to maintain high throughput even with a batch size of one input sample.

Batch-1 RNN calculation is dominated by matrix-vector multiplication (MxV) since there is only one input stream, and the fetched big weight matrix cannot be shared across multiple input streams. The computational cost of this calculation grows quadratically with a linear increase in RNN units. The reasons why it is difficult to achieve low cost, latency, and power simultaneously for RNN hardware inference are fourfold. First, the temporal dependence between the current and previous network output creates a critical path that limits the parallelism between time steps. Second, large networks are essential for high accuracy, leading to a large memory footprint that is expensive to buffer on-chip. Third, MxV is a memory-bounded operation, and the available memory bandwidth limits the minimum latency to fetch the large weight matrices. Fourth, memory access consumes at least 10× more energy than arithmetic operations with the same number of bits [9], [10]. In short, the key to achieving low-cost, low-latency, low-power RNN inference is to reduce the memory bottleneck, that is, to minimize the needed access of weights.

A popular method to reduce memory access is to sparsify the weight matrices. Pruning methods [11]–[13] remove unimportant connections between neurons, resulting in sparse weight matrices with a smaller memory footprint than...
the original dense matrices. Structured pruning was later introduced in RNN accelerators to address the problem of workload (WL) imbalance caused by irregular weight sparsity patterns after pruning [14]–[17]. The bank balanced sparsity (BBS) [15] method divides the rows of a weight matrix into banks of equal length. Then, fine-grain pruning is applied to each bank with an equal number of nonzero elements in each bank. The hierarchical coarse-grain sparsity (HCGS) structured pruning method prepartitions the weight matrix with hierarchical levels of squared submatrices and removes equal numbers of weight elements in each submatrix to ensure a balanced WL. This method was used in a recent power-efficient application-specific integrated circuit (ASIC) RNN accelerator [17]. Structured weight matrices can also reduce the effective number of weights and MxV cost without increasing sparsity: Block-circulant matrices enable the use of the fast Fourier transform to reduce the cost of MxV from \( O(n^2) \) to \( O(n \log(n)) \) [18]–[20].

Another way to reduce memory access requirements is to increase the temporal sparsity of activations, as in spiking neural networks. The delta network (DN) algorithm was inspired by the neuromorphic principle that neurons have sparse activity transmission and introduced as a method to induce temporal sparsity in deep networks by replacing state vectors with delta vectors that contain the deltas of the temporal difference of the states between two adjacent time steps [21]. Zeroing deltas that are below a delta threshold produce sparse delta vectors. Using these sparsified delta vectors results in an insignificant accuracy loss if the RNN is properly trained. Using hardware that can skip zeros in delta state vectors, we can remove operations from entire columns of the weight matrix, which is intrinsically WL balanced [22], [23]. Temporal sparsity is similar to activation sparsity, and to the best of our knowledge, it was only exploited in three previous hardware accelerator works on convolutional neural networks (CNNs) [24], [25] and RNNs [22].

Finally, quantization of weights and states [26] can be combined with the previous sparsity-inducing methods to reduce the model size further. In the basic DN algorithm, columns of weights that are not skipped are still dense. There is an opportunity to further reduce the memory footprint by inducing weight sparsity in a delta RNN. However, it is challenging to implement efficiently because the hardware has to deal with the irregular sparsity pattern in both delta state vectors and weight matrices. Previous RNN hardware accelerators only exploited either spatial sparsity [12], [15], [16], [19], [20] or temporal sparsity [22], [23]. It is challenging to exploit both sparsity types simultaneously because the hardware has to deal with the static sparsity pattern in weights and the dynamic sparsity pattern in states simultaneously. This article describes an accelerator that achieves a further speedup of LSTM RNN inference by exploiting spatio-temporal sparsity in both weights and states of the network, as shown in Fig. 1. The main contributions of this work are given as follows.

1) We extend the delta network algorithm [21] to LSTM to propose delta long short term memory (DeltaLSTM), inducing temporal sparsity in LSTM networks (see Section II).

2) We introduce a structured pruning method called column-balanced targeted dropout (CBTD)\(^1\) that produces a balanced WL among columns of a weight matrix (see Section III-A). CBTD achieves up to 96% and 94% weight sparsity of an LSTM network without accuracy loss, respectively, on the TIMIT and the large-scale Librispeech datasets. By comparison with the hardware-optimized HCGS pruning method [17], CBTD achieves 10\times lower accuracy loss on Librispeech (see Section VI-B).

3) We present the first hardware LSTM-RNN accelerator, Spartus, that exploits both spatial and temporal sparsities in LSTM (see Section IV). The LSTM weight matrices are encoded in our customized sparse matrix format called column-balanced compressed sparse column (CBCSC) that can be efficiently processed by the hardware (see Section III-C).

4) Spartus is reconfigurable in the number of arithmetic processing element (PE) and, thus, can be easily implemented on various sizes of field-programmable gate array (FPGA) (see Section IV-D). Evaluated on the TIMIT dataset, Spartus on the largest Xilinx Zynq FPGA achieves 1-\(\mu\)s inference latency of an LSTM layer with 4.7 million parameters, 9.4-TOP/s effective batch-1 throughput, and 1.1-TOP/J effective power efficiency. In comparison to previous RNN accelerators (see Section VII-A), the Spartus throughput and power efficiency are, respectively, 4\times and 7\times higher than previous state-of-the-art FPGA accelerators BBS [15] and DeltaRNN [22] (see Section VII-A). Edge-Spartus on the smallest Zynq FPGA achieves 33.6-GOP/s/W effective power efficiency, which is 4\times higher than the previous EdgeDRNN [23] accelerator that only exploited temporal sparsity (see Section VII-B), and Edge-Spartus uses inexpensive external memory to run even the largest networks.

II. LSTM AND DELTALSTM NETWORKS

This section introduces the background of the LSTM networks and describes our proposed variant architecture of LSTM called DeltaLSTM to realize efficient LSTM inference.

A. LSTM

An LSTM unit is composed of an input gate \( i \), a forget gate \( f \), a cell gate \( g \), an output gate \( o \), and a memory cell state \( c \). Gates \( i \), \( f \), and \( g \) control the update of the cell \( c \) state. Gate \( o \) determines the proportion of cell memory that is transferred to the hidden state output \( h \). In an LSTM layer, each gate receives two input sequences of length \( T \), including an input sequence \( X = \{x_t|1 \leq t \leq T, t \in \mathbb{N}\} \) and a sequence of previous hidden states \( H_{in} = \{h_t|0 \leq t \leq T - 1, t \in \mathbb{N}\} \) from the unit itself. At each time step, the LSTM layer generates a new hidden state vector, giving a sequence \( H_{out} = \{h_t|1 \leq t \leq T, t \in \mathbb{N}\} \).

\(^1\)https://github.com/gaochangw/DeltaLSTM-CBTD
The formulations of an LSTM layer are given as
\[ i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \]
\[ f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \]
\[ g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \]
\[ o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \]
\[ c_t = f_t \odot c_{t-1} + i_t \odot g_t \]
\[ h_t = o_t \odot \tanh(c_t) \]  \hfill (1)
where \( W \) denotes weight matrices, \( b \) denotes bias vectors, and \( \sigma \) denotes the logistic sigmoid function. The symbol \( \odot \) signifies the pointwise multiplication.

### B. DeltaLSTM

The DeltaLSTM can be understood as follows. Given an input sequence, \( X = \{x_t|1 \leq t \leq T, t \in \mathbb{N}\} \), the output is given by \( Y = \{y_t|1 \leq t \leq T, t \in \mathbb{N}\} \)
\[ y_t = W x_t \]
\[ y_t = W \Delta x_t + y_{t-1} \]  \hfill (2)
where \( \Delta x_t = x_t - x_{t-1} \) is the difference between the input sequence elements from adjacent time steps and is called a delta vector. \( W \) is the matrix of weight connections from the input to the neurons. The delta vector can be sparse if all its elements below a delta threshold \( \Theta \) are set to zero; thus, the term \( W \Delta x \) in (2) becomes a dense matrix-sparse vector multiplication, in which multiply-accumulate (MAC) operations in matrix columns that correspond to zero delta vector elements can be skipped to reduce weight memory access.

The DN algorithm was only studied and implemented as delta gated recurrent unit (DeltaGRU). The DeltaLSTM extends the DN algorithm to LSTM RNNs. Using (2), the LSTM equations (1) are converted to the DeltaLSTM equations following the formulations in
\[ i_t = \sigma(D_{i,t}) = \sigma(W_{ii} \Delta x_t + W_{hi} \Delta h_{t-1} + D_{i,t-1}) \]
\[ f_t = \sigma(D_{f,t}) = \sigma(W_{if} \Delta x_t + W_{hf} \Delta h_{t-1} + D_{f,t-1}) \]
\[ g_t = \tanh(D_{g,t}) = \tanh(W_{ig} \Delta x_t + W_{hg} \Delta h_{t-1} + D_{g,t-1}) \]
\[ o_t = \sigma(D_{o,t}) = \sigma(W_{io} \Delta x_t + W_{ho} \Delta h_{t-1} + D_{o,t-1}) \]
\[ c_t = f_t \odot c_{t-1} + i_t \odot g_t \]
\[ h_t = o_t \odot \tanh(c_t) \]  \hfill (3)
where the terms \( D \) denote the delta memory for each gate, and they are MxV results accumulated over time. The delta memory terms in DeltaLSTM at \( t = 1 \) correspond to the bias terms in the LSTM and are initialized to zeros. Because the delta threshold forces the partial elements of the delta vectors to be zeros, two vectors \( \hat{x}_{t-1} \) and \( \hat{h}_{t-2} \) are used to store the correct previous states to prevent accumulating errors in delta memories. Elements of \( \hat{x}_{t-1} \) and \( \hat{h}_{t-2} \) are updated only when their corresponding delta vector elements are above the delta threshold. The delta vector update process is defined by the following equations:
\[ \Delta x_t = \begin{cases} x_t - \hat{x}_{t-1}, & |x_t - \hat{x}_{t-1}| > \Theta \\ 0, & |x_t - \hat{x}_{t-1}| \leq \Theta \end{cases} \]  \hfill (4)

### III. STRUCTURED PRUNING OF DELTALSTM

The DN algorithm converts dense-matrix-dense-vector multiplication in LSTM to dense-matrix-sparse-vector multiplication to save columns of MxV operations, but the remaining columns are still dense. These dense columns can be sparsified by pruning. However, random access of weight matrix columns is required to exploit temporal sparsity in delta state vectors, and this leads to difficulties in further exploiting sparsity in weight columns, that is, parallelizing MAC operations for hardware PEs on remaining nonzero elements.

Although fine-grain pruning methods [11] could achieve high sparsity around 90% in RNN weights with negligible accuracy loss, they introduce irregular nonzero element distribution in the pruned sparse matrix leading to an unbalanced WL for PEs. Other structured pruning methods, such as BBS, can create structured sparse weights, but the WL will not be balanced once combined with temporal DeltaLSTM. This is because the nonzero values are not evenly distributed across columns and will lead to an unbalanced WL when combined with temporal sparsity. In this work, we propose a hardware-oriented pruning method called CBTD that balances the WL even with the random weight column access in DeltaLSTM.

#### A. Column-Balanced Targeted Dropout

As shown in Fig. 2, interleaved rows of MxV WL are assigned to PEs that have MAC units for MxV computation in the Spartus accelerator. The procedure of applying CBTD on a weight matrix is shown in Algorithm 1. Given \( M \), which is the number of PEs along the column direction in the
Algorithm 1 CBTD

Data: A, matrix;
Q, the number of columns in A;
H, the height of columns in A;
γ, target sparsity;
α, dropout probability;
M, number of PEs allocated along a column.

Result: A sparse weight matrix B of which columns having balanced workload for each PE;
Build set C containing all columns of A, where
C = {c_j | c_j ∈ ℝ^H, 1 ≤ j ≤ Q, j ∈ ℕ};
Shuffle and split columns in set C into subcolumns
S = {s_ij | s_ij ∈ ℝ^{H/M}, 1 ≤ i ≤ M, 1 ≤ j ≤ Q, i, j ∈ ℕ};
for j = 1 to Q do
  for i = 1 to M do
    Sort elements of s_ij by their magnitudes;
    Set the smallest ⌊H/M * γ⌋ elements in s_ij to zero with a probability of α;
  end
end
Reverse the shuffling and splitting to build a sparse matrix B from subcolumns in set S;
return B

Algorithm 2 LSTM Training With CBTD

Data: W, LSTM weight matrices;
Q, the number of columns in W;
H, the height of columns in W;
γ, target sparsity;
α, dropout probability;
Δα, step size of dropout probability;
M, number of PEs per column.

Result: Trained network with a sparse weight matrix in which columns have a balanced workload for each PE;
α = 0;
for iterations do
  Forward Propagation;
  Backward Propagation;
  Update Parameters W;
  CBTD (W, Q, H, γ, α, M);
  if α < 1 then
    α = α + Δα;
  end
end

B. LSTM Training With CBTD

The CBTD is applied to the training procedure of LSTM networks, as described in Algorithm 2. The CBTD is used to set relatively unimportant weights to zeros in each epoch after the parameter update step. Weights that are set to zero in the previous epoch are allowed to recover during the parameter update step in the next epoch. The target sparsity γ is fixed throughout the training process. The dropout probability, α, is increased gradually from 0 to 1 with a step size of Δα, which determines the number of epochs needed for the LSTM weight sparsity to reach the target sparsity γ. This training method guarantees that the network reaches the target sparsity and the same number of nonzero elements between columns or between subcolumns at the end of the training.

Spartus accelerator, CBTD splits each column into the same number of groups (also called subcolumns) as the number of PEs. Thus, M determines the size of each subcolumn for an LSTM layer of a certain size. Next, weight elements in each subcolumn are sorted by their magnitudes. Then, the smallest ⌊H/M * γ⌋ portion of elements in each subcolumn is set to zero with a dropout probability α and a target sparsity, γ. The α probability was used to introduce stochasticity in the targeted dropout process. The same γ and α are used for all subcolumns to ensure the same number of nonzero elements in each subcolumn. Therefore, M determines the granularity of CBTD. With a larger M, the locations of nonzero weights are more tightly constrained due to the smaller subcolumn size and vice versa.
Algorithm 3 CBCSC Format Encoding

Data: $A$, a matrix;
$Q$, the number of columns in $A$;
$H$, the height of columns in $A$;
$M$, the number of PEs in a MAC array;
$\gamma$, target sparsity;

Result: Sparse matrix in CBCSC format stored in value (VAL), local index (LIDX), burst length (BLEN);

Build set $C$ containing all columns of $A$, where $C = \{ c_j | c_j \in \mathbb{R}^H, 1 \leq j \leq Q, j \in \mathbb{N} \}$;
Shuffle and split columns in set $C$ into subcolumns $S[s_j] | s_j \in \mathbb{R}^{H/M}, 1 \leq i \leq M, 1 \leq j \leq Q, i, j \in \mathbb{N}$;
for $j = 1$ to $Q$
  for $i = 1$ to $M$
    for $k = 1$ to $H/M$
      if $s_{ij}[k] \neq 0$
        VAL.append($s_{ij}[k]$)
        LIDX.append($k$)
    end
  end
BLEN = $[H/M \ast (1 - \gamma)]$;

C. Column-Balanced Compressed Sparse Column Format

To fully utilize weight sparsity in an RNN pruned by CBTD, we propose a new sparse weight matrix format method called CBCSC based on the original compressed sparse column (CSC) format [27]. A sparse matrix encoded in CSC has three vectors: VAL for nonzero weight elements, index (IDX) containing the indices of elements in VAL, and column pointer (CP) containing pointers to the start of a new column. The problem of CSC is that the nonzero elements are not arranged in a regular form that benefits PE access. During run-time, the number of weight elements for each PE at the memory interface is different, requiring arbitration between PEs to ensure the correct dispatching of weights. Arbitration reduces the effective memory bandwidth for weight access. To overcome this problem, we propose CBCSC to force the same number of nonzero elements for each PE at the memory interface. The procedure of CBCSC encoding is illustrated in Fig. 3, and the steps are described in the following.

1) Assign interleaved rows to PEs. Columns of the weight matrix are sliced, and interleaved elements in each column are grouped into subcolumns. Each subcolumn is assigned to a single PE.
2) Find the local index of each nonzero element within the subcolumn that it resides in. Zero elements in each subcolumn are discarded, and nonzero elements are aggregated into a dense vector.
3) Encode CBCSC by allocating nonzero values and corresponding indices into two vectors VAL and LIDX. Another scalar value called BLEN indicates the number of nonzero elements in each subcolumn. This process is described in Algorithm 3.

In contrast to CSC, the weight matrix encoded by CBCSC has the same number of nonzero elements in VAL for each PE; thus, arbitration is not needed when fetching weight data and the logic area can be reduced.

IV. ACCELERATOR DESIGN

A. Top-Level Architecture

Fig. 4 shows the top-level architecture of the Spartus accelerator. The accelerator was implemented on the programmable logic (PL) of Xilinx Zynq System-on-a-Chip (SoC), which also has an ARM Cortex-A9 CPU as the host on the processing system (PS) side. Spartus is composed of a controller (CTRL), an state memory (SMEM) block, an input processing unit (IPU), MAC arrays, weight memory (WMEM) banks, adder trees (AT), and an output buffer (OBUF). An Xilinx direct memory access (DMA) IP block controlled by the host is used to manage I/O communications between the accelerator and the host. Input vectors, $x$, are streamed from PS to PL through the DMA module and buffered in the SMEM block to hide the transfer latency. The SMEM block is also used to buffer LSTM activations $h$. The IPU concatenates $x$ and $h$ to compute the delta state vectors $\Delta s$. The delta state vectors are encoded into nonzero value (NZV) and nonzero index (NZI). NZIs are dispatched to CTRL to generate the physical memory addresses of weights in each WMEM bank, and NZVs are dispatched to MAC arrays to be multiplied with the fetched weights. There are $N$ MAC arrays. Each of the first $N - 1$ arrays contains $M$ PEs that perform $M\times V$ between NZVs and corresponding weight columns. The last array has $M$ heterogeneous processing element (HPE) that is also responsible for post-$M\times V$ activation generation. The OBUF helps to hide the latency of transferring the last layer’s activations to the host.

B. Input Processing Unit (IPU)

The IPU computes delta state vectors $\Delta s_t$ from both input vectors $x_t$ and hidden layer activations $h_t$. It then generates NZV and NZI that, respectively, contain the nonzero values of $\Delta s_t$ and their corresponding indices. As shown in Fig. 5, inputs of the IPU are streamed in from the SMEM block.
In SMEM, the lengths of $x_t$ and $h_t$ vectors are zero-padded to the length that is a multiple of $M$ and concatenated into a single state vector $s_t$. During the inference, the IPU receives $M$ elements of $s_t$ per clock cycle whenever the state first-in first-out (FIFO) (S-FIFO) is not full. The state vector $s_t[M − 1 : 0]$ is then partitioned into $N$ equal segments $s_t[M/N − 1 : 0]$, each of which is fed into a DPE, as shown in Fig. 6. Details of the state vector partition will be discussed in Section V-E.

C. Delta Processing Unit (DPE)

Fig. 6 shows the architecture of the DPE. Following (4)–(7), the DPE calculates a delta vector $\Delta s_t$ from $s_t$ and $s_{t−1}$. Each DPE receives a partitioned $s_t$ segment of length $I = M/N$, given that $M$ must be divisible by $N$. The input elements of a DPE are denoted as $s = [s[i]|0 ≤ i ≤ I − 1, i \in \mathbb{N}]$. $s_t$ is buffered in the IPU input FIFO, and $s_{t−1}$ is stored in the lookup table-based memory [lookup table-based random access memory (LUTRAM)] blocks in the DPE. Each LUTRAM block is addressed by a counter (CNT), which is incremented by one when valid is asserted. Equations (4) and (6) are implemented as the threshold function (TF) block, while (5) and (7) are realized by controlling the write enable (we[$i$]) signal of the LUTRAM using the output of a comparator that produces a high logic state when $|s_t|$ is larger than the delta threshold $\Theta$. The first nonzero element in $\Delta s_t$ with its index $i$ is selected as NZV and NZI, respectively, by two multiplexers controlled by a decoder (DEC) according to signal code[$M/N − 1:0$]. code = comp & mask, where elements of mask are initialized as ones. Controlled by the signal en[$i$], when mask[$i$] = 1 and comp[$i$] = 1 in the current cycle, mask[$i$] is overridden by zero in the next clock cycle. In this way, mask is used to disable nonzero $\Delta s_t$ once it is already selected and added to the NZV. The NZVs and NZIs generated from DPEs are buffered in their corresponding delta state FIFO (D-FIFO), which drives the input of PEs or HPEs in MAC arrays.

D. Multiply–Accumulate Arrays

sparse matrix-sparse vector multiplication (SPMxSPV) in Spartus is handled by the $N$ MAC arrays. Each array receives NZVs from its corresponding DPE in the IPU. In Fig. 4, the leftmost $N − 1$ MAC arrays have $M$ PEs, and the rightmost MAC array has $M$ HPEs. As shown in Fig. 7, a PE has a MAC unit synthesized by a digital signal processing (DSP) block that performs up to 16-bit $\times$ 16-bit multiplication and 48-bit accumulation between NZVs and weights. In addition to the DSP block, the HPE shown in Fig. 8 has multiplexers before each input operand of the DSP to reuse for pointwise multiplication and addition. Furthermore, the HPE also has tanh and sigmoid blocks implemented by lookup tables. Each PE or HPE has a dedicated LUTRAM block to buffer their corresponding partial sums. The LUTRAM is addressed by the IDX of weights encoded in the CBCSC format.

After SpMxSPV, delta memory terms $DM$, as defined in (3), are obtained by accumulating partial sums of all MAC arrays by $M$ adder trees. Outputs of the adder trees are then fed to HPEs throughout the port AT for activation generation. Activation $h_t$ is first stored in SMEM for delta state vector computation of the next time step and then streamed out to the host through DMA.

E. Network Adaptation

The Spartus accelerator supports fixed-point weights and activations. To run a DeltaLSTM network on Spartus, the
Fig. 9. Sparse DeltaLSTM weights are encoded into CBCSC format and then split into WMEM banks for each MAC array to access independently. We show an example with \( N = 2 \) MAC arrays and \( M = 2 \) PEs/HPEs per array.

TABLE I

| FPGA          | DSP   | BRAM/M20K | LUT/ALM | FF  | Process | Cost | RNN Accelerator |
|---------------|-------|-----------|---------|-----|---------|------|-----------------|
| XC7Z007S      | 60    | 50 (1.3 Mb)| 14,400  | 555 | $3.978  | $17.926 |
| XC7Z100       | 2,020 | 755 (26.5 Mb)| 277,400 | 28nm | $17.926 |
| XCKU060       | 2,760 | 1,080 (38.9 Mb)| 331,680 | 28nm | $3.978  |
| SX660         | 3,376 | 2,133 (41.7 Mb)| 250,540 | 28nm | N/A     |
| XC7VX690T     | 3,600 | 1,470 (51.7 Mb)| 433,200 | 28nm | $3.978  |
| VX1570        | 3,036 | 2,713 (53.0 Mb)| 427,200 | 28nm | $3.978  |

1 Single BRAM Size = 36 Kb. Single M20K size = 20 Kb.  
2 The look-up table (LUT) in Xilinx FPGAs [28] is not equivalent to the Adaptive Logic Module (ALM) in Intel FPGAs [29].  
3 Costs extracted from Digi-Key US in March 2022.

The network should be trained starting with floating-point parameters and quantized to fixed-point numbers before inference. To run an LSTM network with Spartus, weight matrices \( W_{ii}, W_{hi}, W_{hf}, W_{ig}, W_{hg}, W_{io}, \) and \( W_{ho} \) are stacked to a single matrix \( W_s \) given as follows:

\[
W_s = \begin{bmatrix}
W_{ii} & W_{hi} \\
W_{ig} & W_{hg} \\
W_{if} & W_{hf} \\
W_{io} & W_{ho}
\end{bmatrix}
\]  \( (8) \)

During matrix-vector multiplication, the stacked weight matrix is multiplied by the delta state vector \( \Delta s_t \), which follows the same partition pattern as the state vector \( s_t \). Accordingly, \( W_s \) is partitioned into \( N \) submatrices. Each submatrix only contains columns that will be multiplied by NZVs from its corresponding DPE and are encoded into the CBCSC format, as shown in Fig. 9 and discussed in Section III-C. Then, the interleaved columns of the CBCSC weight are split into \( N \) submatrices. Each submatrix is stored in a WMEM bank dedicated to its corresponding MAC array to avoid bank conflict. The delta vector partition pattern follows how the stacked weight matrix is split. The network weights are quantized to 8 bits, while 8- and 10-bit LIDX are used for Spartus and Edge-Spartus, respectively, in the CBCSC format.

V. EXPERIMENTAL SETUP

A. Hardware Implementation

To demonstrate the scalability of Spartus, we evaluated its hardware performance on two configurations with different numbers of PEs, Spartus and Edge-Spartus, both of which are synthesized in Vivado 2018.2. Spartus is implemented on an AVNET Zynq XC7Z100 with a custom baseboard for Spartus. Right: AVNET MiniZed XC7Z007S for Edge-Spartus. See Table I.

Fig. 10. FPGA development boards used in this work. Left: AVNET Zynq XC7Z100 with a custom baseboard for Spartus. Right: AVNET MiniZed XC7Z007S for Edge-Spartus. See Table I.
BRAM capacity is too small to buffer the large network; thus, the program initializes weights in off-chip DDR3L DRAM, and the accelerator fetches weights from the high performance (HP) AXI4 slave port interfacing PL and PS.

B. Feature Extraction and Network Setup

We evaluate the impact on the accuracy of both the CBTD method and the DeltaLSTM algorithm on a speech recognition task using the TIMIT and larger Librispeech datasets.

The TIMIT dataset [30] has a standard 462 speaker training set with all dialect sentences (SA) removed following [4]. The development (DEV) and test (TEST) sets have utterances of 50 and 24 speakers, respectively. The 123-D features are extracted from the input and consist of the 40 coefficients of FFT-based filter banks distributed on a Mel-scale, plus the energy term, and their first- and second-order temporal derivatives [4]. The connectionist temporal classification (CTC) loss function [31] is used during training so that the final logit layer of the acoustic model (AM) generates phonemes directly without requiring a sophisticated decoding algorithm. The phone error rate (PER) results were collected using a simple greedy decoder that selects the index of the phoneme class with the highest score.

The large-scale LibriSpeech dataset has 1000 h of audio-book speech based on LibriVox’s library [32]. For Librispeech, we followed the same AM training process leading to the LSTM baseline in the PyTorch-Kaldi framework using the 100-h subset [33]. We used the Kaldi toolkit [34] to extract 40-D feature-space maximum likelihood linear regression (fMLLR) features [35] and decode the AM output using a Viterbi decoder with a 4-gram language model to calculate the word error rate (WER). The PyTorch-Kaldi [33] framework was used to train the AM. We used these training and evaluation processes to have a fair comparison with the PyTorch-Kaldi baseline and the HCGS [17] method.

We trained LSTM-AMs on both datasets. The last LSTM layer is followed by a fully connected layer (FCL) having the same number of units and a final logit layer. The networks implemented on the Spartus hardware accelerators are quantized during training to 8-bit weights and 16-bit activations using the dual-copy rounding method [36].

C. Pretrain and Retrain

We adopted a pretrain–retrain process for this task to train networks with spatio-temporal sparsity. For the TIMIT dataset, during the pretrain phase, the LSTM layers were initialized and trained with the CBTD applied for 150 epochs. We set the step size of dropout probability $\Delta a$ to 1/30 so that the target sparsity $\gamma$ was achieved within the first 30 epochs and maintained during the remaining epochs. For the Librispeech dataset, we pretrain the networks for 24 epochs (the same as the PyTorch Kaldi baseline) and anneal the step size of dropout probability of CBTD during the first 10 epochs. The CBTD was also applied to the FCL with the same $\Delta a$ and $\gamma$. The pretraining process was early stopped at the epoch that achieved the best accuracy on the development set.

During the retraining phase, weights of the pretrained LSTM layers are copied into DeltaLSTM layers of the same size to be retrained for 50 epochs for the TIMIT dataset and 5 epochs for the Librispeech dataset with $a = 1$. The retrain phase was also early stopped at the best accuracy on the development set. The final accuracy results for the TIMIT dataset are reported on the core test set, which has 192 samples. For Librispeech, the final WER results were evaluated on the standard test set.

VI. RESULTS

A. Design Space Exploration

We determine how the numbers of PEs and MAC arrays, and the delta threshold $\Theta$ impact the accuracy and hardware performance as evaluated on TIMIT.

1) Pretrain: Phone Error Rate Versus Weight Sparsity: This section reports the impact of CBTD on weight sparsity and accuracy. Fig. 11(a) and (b) shows the evolution of PER on the development set (DEV-PER) over the target sparsity. Each data point in Fig. 11 is the average value over five runs. We explored using different numbers $M$ of PEs per MAC array with values of $M = 32/64/128$ for the CBTD method. The horizontal dashed line shows the DEV-PER of the network with 32-bit floating-point (FP32) parameters trained without using CBTD and quantization, which achieved 20.30% DEV-PER.

Results show that the accuracy with different numbers of PEs is similar. Most of the DEV-PER values of LSTM networks trained with CBTD were worse than the FP32 baseline when the target sparsity is below 40%. Similar to the dropout [37] and targeted dropout [13] regularization methods, CBTD helped regularize the LSTM network. Target sparsity $\gamma$ between 50% and 90% resulted in better DEV-PER than FP32 due to the regularization effect of CBTD. A similar regularization effect on the TIMIT dataset due to weight pruning on LSTM was also observed in [15]. The best PER results are achieved when $\gamma$ is between 90% and 94%. The pretrained networks with CBTD achieve up to 96% weight sparsity without loss of accuracy compared to the FP32 baseline results. To achieve the best PER while having speedup as high as possible, the network having 94% weights sparsity was used in the retrain phase, which achieves a DEV-PER value of around 19.30% among the best PER values of all sparsity levels.

2) Retrain: Phone Error Rate Versus Delta Threshold: The purpose of the retrain phase is to induce temporal sparsity using the DeltaLSTM model. Temporal sparsity introduces zero elements that can be skipped to save the computations and the memory access but can lead to accuracy degradation once the sparsity is higher than a certain value. This section reports the results of the retrain phase and explores the optimal condition to achieve a balance between speedup and accuracy.

Fig. 11(c) shows the DEV-PER of the network retrained with DeltaLSTM layers. During the retraining phase, various delta thresholds $\Theta$ were used. The same thresholds were applied to all LSTM layers. The CBTD was still used during the whole training phase with a step size of dropout rate $a$ fixed to 1. Results show that DEV-PER increases with
increased delta thresholds $\Theta$, and the network trained with $M = 64$ achieves the best DEV-PER, which is lower than the FP32 baseline until $\Theta > 0.3$. Thus, we set $\Theta = 0.3$ to evaluate the performance of the accelerator in Section VI.

3) Workload Balance Between MAC Arrays: The theoretical peak throughput $v_{\text{peak}}$ of the accelerator on the PL is given as

$$v_{\text{peak}} = 2 \cdot f_{\text{pl}} \cdot K$$

where $f_{\text{pl}}$ is the operation clock frequency of the PL and $K$ is the total number of MAC units. In Spartus, $K = M \times N$, where $N$ is the number of MAC arrays. Thus, the theoretical peak throughput is proportional to the number of MAC arrays; however, the actual hardware throughput is affected by the WL imbalance between MAC arrays.

Unlike the structured sparse weight matrix induced by CBTD, the sparsity pattern of the delta state vector $s_t$ is dynamically updated during LSTM inference in each time step. Therefore, partitioned $\Delta s_t$ vector segments, which are encoded as NZVs by DPEs, are likely to have different numbers of nonzero values in each time step of the LSTM computation. Provided that there are $N$ MAC arrays in the accelerator, the WL, which is the number of nonzero elements of $\Delta s_t$ allocated to the $n$th MAC array at time step $t$, is $WL_{t,n}$. Then, the balance ratio (BR) of MAC arrays is given as

$$BR = \frac{\text{WL}_{t,\text{mean}}}{\text{WL}_{t,\text{max}}}$$

$$WL_{t,\text{mean}} = \frac{1}{N} \sum_{i=1}^{N} WL_{t,i}$$

$$WL_{t,\text{max}} = \max(WL_{t,1}, WL_{t,2}, \ldots, WL_{t,N})$$

where $WL_{t,\text{mean}}$ and $WL_{t,\text{max}}$ are, respectively, the mean and max WL of all MAC arrays at time step $t$. The BR is obtained by running the hardware on a temporal sequence with a length of $T$ and getting the sum of $WL_{t,\text{mean}}$ and $WL_{t,\text{max}}$ over $T$ time steps. The performance of hardware is bottlenecked by the max WL $WL_{t,\text{max}}$ at each time $t$. The optimal WL balance can be achieved once the WL of each MAC array equals to $WL_{t,\text{mean}}$. Therefore, the closer BR is to 1, the more balanced the WL between MAC arrays.

Fig. 12 shows the BR values of MAC arrays evaluated on all samples in the core test set of TIMIT running the best DeltaLSTM network selected after the retraining phase. Increasing delta threshold $\Theta$ or the number of PEs resulted in a more imbalanced allocation of nonzero elements between MAC arrays. To maximize the hardware performance at $\Theta = 0.3$, we set the number of MAC arrays between $N = 8$ and $N = 16$, where BR is between 0.8 and 0.69. Thus, the loss of hardware performance due to WL imbalance is 20% to 31%, which is small compared to the 8×16×16 peak throughput gain by increasing $N$ from 1 to 8 and 16. However, using $N > 8$ resulted in routing congestion. Therefore, we set $M = 64$ MAC arrays and $N = 8$ PEs for our final Spartus implementation. For Edge-Spartus, we set $M = 4$ and $N = 1$ to have a similar off-chip DRAM bandwidth as the EdgeDRNN accelerator (EdgeDRNN requires 64 bits per clock cycle, while Edge-Spartus requires 72 bits per clock cycle) for a relatively fair comparison in Sections VII-A and VII-B.

B. Model Accuracy

We evaluated the final accuracy of our AM using our proposed training and optimization methods on the TIMIT
TABLE II

ACCURACY RESULTS ON THE CORE TEST SET OF TIMIT WITH DIFFERENT SIZES OF LSTM NETWORKS AND OPTIMIZATION METHODS. (L DENOTES THE NUMBER OF LSTM LAYERS; H DENOTES THE NUMBER OF UNITS IN EACH LSTM LAYER; UNI DENOTES THAT THE NETWORK IS UNIDIRECTIONAL; THE BOLD NETWORK NAMES ARE SUPPORTED BY SPARTUS; THE BOLD ROWS HAVE THE HIGHEST ARITHMETIC OPERATIONS SAVING WITHOUT LOSING ACCURACY; AND RESULTS WERE AVERAGED FROM FIVE RUNS)

| Network | γ | Θ | Weight Precision | Model Size (MB) | Weight Sparsity (%) | Temporal Sparsity (%) | TEST-PER (%) | Accuracy Improvement (%) | Arithmetic Operations Saving |
|---------|---|---|-----------------|----------------|---------------------|----------------------|--------------|--------------------------|-----------------------------|
| LSTM-3L-512H-UNI | - | - | FP32 | 23.18 | 0 | 0 | 22.7±0.16 | 0 | 1x |
|         | 0.80 | - | INT8 | 1.27 | 78.13 | 0 | 21.5±0.12 | 3.4 | 4.6x |
|         | 0.90 | - | INT8 | 0.72 | 87.50 | 0 | 21.3±0.34 | 1.4 | 8.0x |
|         | 0.97 | - | INT8 | 0.18 | 96.88 | 0 | 25.7±0.56 | 2.9 | 32.1x |
| LSTM-3L-768H-UNI | - | - | FP32 | 30.88 | 0 | 0 | 23.2±0.36 | 0 | 1x |
|         | 0.80 | - | INT8 | 8.10 | 0 | 0 | 23.4±0.31 | 0.2 | 1x |
|         | 0.90 | - | INT8 | 1.69 | 79.17 | 0 | 27.2±0.27 | 1.5 | 4.8x |
|         | 0.94 | - | INT8 | 0.84 | 89.58 | 0 | 21.4±0.15 | 2.0 | 9.6x |
|         | 0.97 | - | INT8 | 0.51 | 93.75 | 0 | 22.4±0.96 | 0.8 | 16.0x |
| LSTM-2L-1024H-UNI | - | - | FP32 | 56.81 | 0 | 0 | 22.3±0.29 | 0 | 1x |
|         | 0.80 | - | INT8 | 14.20 | 0 | 0 | 22.0±0.29 | 0.3 | 1x |
|         | 0.90 | - | INT8 | 2.88 | 79.69 | 0 | 21.1±0.31 | 0.9 | 4.1x |
|         | 0.94 | - | INT8 | 1.55 | 89.06 | 0 | 20.8±0.33 | 1.5 | 9.1x |
|         | 0.97 | - | INT8 | 0.44 | 96.88 | 0 | 22.7±0.39 | 0.4 | 32.1x |
| DeltaLSTM-2L-1024H-UNI-CTBD (w/ Spatio-Temporal Sparsity) | | | | | | | | | |
|         | 0.94 | 0.3 | INT8 | 0.89 | 93.75 | 74.22 | 20.6±0.30 | 1.7 | 62.1x |

TABLE III

ACCURACY RESULTS ON THE TEST SET OF LIBRISPEECH WITH DIFFERENT SIZES OF LSTM NETWORKS AND OPTIMIZATION METHODS TRAINED ON THE 100-h TRAINING SET. (BI DENOTES THAT THE NETWORK IS BIDIRECTIONAL; ACCURACY IMPROVEMENT IS THE WER DIFFERENCE TO THE BASELINE FLOATING-POINT CONVENTIONAL RNN)

| Network | γ | Θ | Weight Precision | Model Size (MB) | Weight Sparsity (%) | Temporal Sparsity (%) | TEST-WER (%) | Accuracy Improvement (%) | Arithmetic Operations Saving |
|---------|---|---|-----------------|----------------|---------------------|----------------------|--------------|--------------------------|-----------------------------|
| LSTM-4L-512H-UNI (PyTorch-Kaldi Baseline [33]) | - | - | FP32 | 14.84 | 0 | 0 | 6.4 | - | 1x |
| LSTM-4L-1024H-UNI-CTBD (w/ Spatial Sparsity) | 0.90 | - | INT8 | 0.46 | 87.5 | 0 | 6.2 | 0.2 | 8.0x |
|         | 0.94 | - | INT8 | 0.23 | 93.75 | 0 | 7.1 | -0.7 | 16.0x |
| LSTM-3L-512H-UNI | - | - | FP32 | 21.3 | 0 | 0 | 7.0 | - | 1x |
| LSTM-3L-512H-UNI-BIGS (w/ Spatial Sparsity) | - | - | INT6 | 0.29 | 93.75 | 0 | 20.8 | -4.4 | 16.0x |
| LSTM-3L-512H-UNI-CTBD (w/ Spatial Sparsity) | 0.94 | - | INT8 | 0.33 | 93.75 | 0 | 7.3 | -0.3 | 16.0x |
| LSTM-2L-1024H-UNI | - | - | FP32 | 56.81 | 0 | 0 | 8.0 | - | 1x |
|         | 0.80 | - | INT8 | 14.20 | 0 | 0 | 7.6 | 0.4 | 1x |
|         | 0.90 | - | INT8 | 2.88 | 79.69 | 0 | 6.7 | 1.3 | 4.9x |
|         | 0.94 | - | INT8 | 1.55 | 89.06 | 0 | 6.5 | 1.5 | 9.1x |
|         | 0.97 | - | INT8 | 0.44 | 96.88 | 0 | 8.3 | -0.3 | 32.1x |
| DeltaLSTM-2L-1024H-UNI-CTBD (w/ Spatio-Temporal Sparsity) | | | | | | | | | |
|         | 0.94 | 0.05 | INT8 | 0.89 | 93.75 | 80.45 | 7.4 | 0.6 | 81.8x |

and Librispeech test sets with M = 64 and different delta thresholds.

1) TIMIT: To demonstrate the effectiveness of CBTD, we trained the AMs with various numbers of layers and layer sizes on the TIMIT dataset, as shown in Table II. The largest network has two unidirectional LSTM layers with 1024 units per layer, which is the same as the networks used in ESE [12] and other related previous works, including C-LSTM [19], E-RNN [20], BBS [15], and E-LSTM [16]. We observe that networks pretrained with our CBTD method result in better PER on the test set than the corresponding FP32 baselines when the spatial (weight) sparsity is roughly between 80% and 94%. Networks are likely to have worse PER when the spatial sparsity is ≥97%. These results on the TIMIT test set are consistent with those shown in Fig. 11 on the development set.

In short, our CBTD method achieves 16× lossless compression of the various sizes of AMs. The previous works either could not achieve the same level of model compression rate or had worse accuracy loss. By further adding temporal sparsity on top of spatial sparsity, our spatio-temporal DeltaLSTM network achieves 170× saving of arithmetic operations and memory access of the MxV in the LSTM layers, which is significantly better than previous works. Fig. 13(a) shows the impact of different delta threshold values on the temporal sparsity evaluated on the TIMIT core test set, and the values are the average temporal sparsity over the two DeltaLSTM layers. Overall, the sparsity of delta input state vector Δx is lower than delta hidden state vector Δh. This is due to
With small delta thresholds, the DeltaLSTM shows a slight regularization effect that helped to achieve the best TEST-PER at 20.63%. With \( \Theta = 0.3 \), the TEST-PER is 21.75%, which corresponds to 0.55% accuracy improvement compared to the FP32 baseline.

2) Librispeech: Beyond the simpler TIMIT dataset, we also evaluate the effectiveness of cost-saving and accuracy impact of our methods on the large-scale corpus, Librispeech, which has not been evaluated in previous FPGA RNN accelerators. Results are shown in Table III. We first applied CBTD on the bidirectional LSTM used in the PyTorch-Kaldi baseline [33] and obtained \( 8 \times \) arithmetic operations saving with 0.2% better WER than the baseline. \( 16 \times \) saving is achievable with 0.7% accuracy loss. The bidirectional LSTM networks achieve the highest accuracy but are not usable in real-time speech recognition systems because they must receive the whole sentence to recognize a single frame. Thus, we are more interested in evaluating our methods on unidirectional LSTM networks. To the best of our knowledge, HCGS [17] is the only previous structured pruning method evaluated on Librispeech. However, the sparse weight pattern of HCGS is predefined at the network initialization stage and maintained throughout training with only changeable weight magnitudes, greatly limiting its ability to find subnets with good accuracy, as discussed in The Lottery Ticket Hypothesis [38], [39]. The CBTD allows both removal and addition of connections during training with only constraints on the distribution of nonzero values across predefined submatrices, which could lead to higher accuracy with the same level of model compression. We applied CBTD on a unidirectional LSTM network of the same size, as reported in [17]. With the same \( 16 \times \) saving in arithmetic operations, CBTD has only 0.3% higher WER, while HCGS caused \( 10 \times \) higher accuracy loss. Finally, we applied our methods to the AM model used for hardware evaluation. We found that the combination of INT8 quantization, CBTD, and DeltaLSTM achieves \( 64 \times \) model compression and \( 81.8 \times \) arithmetic operations saving compared to the corresponding FP32 baseline. Another important observation is that networks pruned by CBTD during training with conservative pruning rates (less or equal to 94% for TIMIT and 90% for Librispeech) all led to better accuracy, proving that CBTD well preserves the regularization effect of pruning, as suggested by related works [38]–[40].

C. Hardware Performance: Throughput and Latency

Since we focus on achieving low latency LSTM inference, in this work, we evaluate the hardware performance with a batch size of 1. The test network is the top layer of our biggest AM having 1024 hidden units, which is the same as previous state-of-the-art FPGA RNN accelerators [12], [15], [16], [19], [20].

Table IV summarized the performance of Spartus and Edge-Spartus with progressive levels of optimizations. With \( M \times N = 512 \) MAC units, the theoretical baseline performance of Spartus is only 0.2 TOP/s, calculated using (9). With the highest level of optimization, Spartus can finish the inference of the big DeltaLSTM layer within 1 \( \mu s \), corresponding to 9.4-TOp/s effective throughput and 46\( \times \) speedup versus...
the baseline. Fig. 13(c) further shows how much the spatial and temporal sparsities contribute to the total speedup. By applying only CBTD to the LSTM network to induce 94% spatio sparsity (16× pruning rate), the Spartus achieves 2.8-TOP/s batch-1 throughput, corresponding to a 14X spatial gain on top of the baseline. After retraining the sparse LSTM layers as DeltaLSTM, further speedup was achieved. With a zero delta threshold, the accelerator achieves 3.6-TOP/s batch-1 throughput. By increasing the delta threshold to 0.3, which is the highest value with better TEST-PER over FP32 baseline on TIMIT, Spartus achieves 9.4-TOP/s batch-1 throughput, which is another 3.3× temporal gain on top of spatial gain. Overall, by combining the 14× spatial gain and 3.3× temporal gain, Spartus achieves 46× speedup by exploiting spatio-temporal sparsity.

Table IV also reports the performance of Edge-Spartus, which was implemented on the FPGA of the smallest Zynq SoC, which has only 0.2-MB on-chip memory and cannot buffer the networks on-chip even after compression. SoC, which has only 0.2-MB on-chip memory and cannot buffer the networks on-chip even after compression. By relying solely on off-chip DDR3L memory bandwidth to fetch the network weights, Edge-Spartus achieves 121.7 μs latency, which is still orders of magnitude faster than the normal 10-ms frame shift used in real-time speech recognition systems [3], [41].

VII. DISCUSSION

A. Comparing Spartus With Previous Works on TIMIT

ESE was the first RNN accelerator that adopted weight pruning to speed up LSTM inference. However, ESE was designed for throughput-oriented inference using very large batch sizes. The performance of ESE was tested with a batch size of 32. DeltaRNN exploited temporal sparsity to accelerate GRU RNNs, but the test network was small compared to other works. C-LSTM and E-RNN used a structured weight matrix with an FFT-based computing mechanism to reduce operations during inference. Table V provides the performance of C-LSTM and E-RNN with the 16× compression ratio reported in the original papers. E-LSTM and BBS adopted structured pruning methods to achieve fine-grained WL balance compared to ESE, and their performance was evaluated with batch sizes of 8 and 1, respectively. BBS achieved the best batch-1 throughput and latency among all previous RNN accelerators. Since we focus on achieving low-latency LSTM inference, for a fair comparison, we computed the performance of ESE and E-LSTM with a batch size of 1 in Table V. Power numbers of Spartus and other accelerators are measured using the wall-plug power of the FPGA board.

We include “latency” and “frame per second (FPS)” to represent the performance of accelerators on their own test networks. Spartus achieves the highest FPS and lowest inference latency among all platforms. However, the test network sizes are different across platforms. Spartus and E-LSTM used test networks with normal LSTM units, while ESE, C-LSTM, E-RNN, and BBS used a Google LSTM network with the same number of neurons because their neuron model has peephole connections and projection layers that shrink the dimension of both recurrent and output connections [42], leading to an overall decrease in the number of network parameters. Thus, we include “effective throughput” and “power efficiency” metrics that take into the number of total operations for computing these networks in the comparison across these accelerators. Compared to DeltaRNN, which also exploited temporal sparsity, Spartus achieves 8× higher effective throughput and 7× higher power efficiency. Compared to C-LSTM and E-RNN that achieved the same 16× weight compression ratio, Spartus achieves around 10× higher FPS and 8× lower latency. By exploiting spatio-temporal sparsity, Spartus achieves 4× higher batch-1 effective throughput than BBS, which was the state-of-the-art accelerator with the highest batch-1 effective throughput.

Table V also includes the resource utilization and names of the FPGA chips. Table I compares FPGAs. A larger FPGA can buffer larger networks on-chip to provide sufficient memory bandwidth to more physical MAC units. New process technologies can help the same accelerator architecture to achieve higher clock frequencies after implementation. It can be observed that Spartus achieves higher performance than previous accelerators even on a smaller FPGA with the earlier 28-nm process, except for DeltaRNN that used the same XC7Z100 FPGA.

B. Comparing Edge-Spartus to EdgeDRNN and ESE

Most previous accelerators achieve their high batch-1 throughput by storing the RNN weights completely on-chip, which is not practical in edge applications using resource-constrained hardware platforms. EdgeDRNN [23] and ESE [12] are the only two previous FPGA RNN accelerators that reported real performance numbers with the off-chip memory bandwidth bottleneck. Thanks to the scalable architecture of Spartus, we could fit Edge-Spartus with four PEs on the tiny FPGA in the Zynq XC7007S SoC. Edge-Spartus uses INT8 weights with a 10-bit local index for each weight; thus, the off-chip memory interface bit width is 72 bits, while the previous EdgeDRNN with eight PEs used INT8 weights and require a 64-bit off-chip memory interface. As shown in Table VI, with the same DDR3L off-chip memory as EdgeDRNN on the MiniZed board, Edge-Spartus achieves 77.3-GOP/s batch-1 effective throughput, which is around 4× higher than EdgeDRNN and close to that of ESE, which uses 8× more MAC units on an FPGA that costs 70× more.

C. DRAM Power Efficiency

On-chip memory fetch consumes over 10× higher energy than arithmetic operations with the same numbers of bits and
TABLE V
COMPARISON OF SPARTUS WITH PRIOR STATE-OF-THE-ART RNN ACCELERATORS ON FPGA (BATCH SIZE = 1)

|                         | ESE [12] | DeltaRNN [22] | C-LSTM [19] | E-RNN [20] | BBS [15] | E-LSTM [16] | Spartus (Ours) |
|-------------------------|----------|---------------|-------------|------------|----------|-------------|----------------|
| #Parameters (M)         | 3.25     | 0.20          | 3.25        | 3.25       | 3.25     | 4.82        | 4.70           |
| Compressed #Parameters  | 0.36     | -             | 0.20        | 0.41       | 0.60     | 0.29        |
| Bit Precision (Act/Wg)  | INT16/12/4 | INT16/16/0   | INT16/16/0  | INT16/16/0 | INT16/16/4| INT8/8/4    | INT16/8/8      |
| Weight Sparsity (%)     | 88.78    | 0             | 0           | 93.75      | 67.5     | 93.75       |
| Act/Temp. Sparsity (%)  | 0        | 92.5          | 0           | 82.56      |          |
| PER on TIMIT (%)        | 20.7     | -             | 24.6        | 20.3       | 32.1     | 21.8±0.3    |
| Language Model?         | Yes      | -             | Unreported  | No         | No       |
| FPGA Platform           | XC7U060  | XC7Z100       | 7V3         | XC7VX690T  | GX150    | SX660       | XC7Z100        |
| DSP (%)                 | 48.3     | 38.0          | 74.3        | 39.6       | 100      | 13          |
| BRAM/M20K (%)           | 87.7     | 60.6          | 65.7        | 65.7       | 92       | 32.1        |
| LUT/ALM (%)             | 88.6     | 54.2          | 58.7        | 59.4       | 68       | 49.2        |
| FF (%)                  | 68.3     | 21.3          | 46.5        | 55.3       | N/A      | 15.6        |
| Frequency (MHz)         | 200      | 125           | 128         | 4096       | 128      | 512         |

TABLE VI
COMPARISON OF EDGE-SPARTUS WITH EDGE-DRNN (BATCH SIZE = 1)

|                        | Edge-DRNN [23] | Edge-Spartus (Ours) |
|------------------------|----------------|---------------------|
| #MACs                  | 32             | 4.7                 |
| Compressed #Parameters  | 0.23           | 0.29                |
| Bit Precision (Act/Wg) | INT16/16/0     | INT16/8/10          |
| Weight Sparsity (%)    | 0              | 93.75               |
| Act/Temp. Sparsity (%) | 90.01          | 82,56               |
| FPGA Platform          | XC7Z007S       |                     |
| DSP (%)                | 13.6           | 7.6                 |
| BRAM (%)               | 66.0           | 76.0                |
| LUT(%)                 | 65.2           | 78.5                |
| FF (%)                 | 34.1           | 41.5                |
| Frequency (MHz)        | 125            |                     |

VIII. CONCLUSION
We proposed Spartus, the first LSTM accelerator that exploits spatio-temporal sparsity to enable the lowest latency

can be over 1000× when using off-chip DRAM [9], [10]. For a memory-bounded algorithm, such as LSTM, the key to enhancing power efficiency is to reduce the amount of memory access per inference. Using Edge-Spartus, we analyzed the potential reduction of energy consumption per frame using various types of DRAM (see Table VII) and showed the results in Fig. 14. By exploiting spatio-temporal sparsity, the DRAM access energy can be reduced by 91.7×. However, this reduction of energy does not completely translate to the power efficiency of the whole system, of which other peripheral modules can also be an important source of power consumption. For example, our XC7Z100 SOM lacks a usable PS sleep mode and, thus, consumes 1.8 W even at idle. Thus, although the power efficiency obtained using wall power reflects the real-world performance of different accelerators, they are biased by the boards that they are implemented on. Our comparison between EdgeDRNN and Edge-Spartus showcases how much improvement of power efficiency can be achieved on the same platform by exploiting spatio-temporal sparsity, and Edge-Spartus achieves roughly 4× higher core and system power efficiency than EdgeDRNN.

TABLE VII
OFF-CHIP DRAM ACCESS ENERGY

| DRAM Type  | DDR3 | DDR3L | GDDR6 | HBM2 |
|------------|------|-------|-------|------|
| Access Energy/Bit (pJ) | 20.3 | 16.5  | 5.5   | 3.9  |

1 The DDR3L (1.35 V) access energy is estimated by scaling down the number of DDR3 (1.5 V) according to their supply voltages.

Fig. 14. Estimated off-chip DRAM access energy of Edge-Spartus per inference (frame).

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
and the highest power efficiency in RNN inference compared to previous work. The spatial weight sparsity was implemented using our newly proposed structured CBTD pruning method. CBTD achieves a comparable compression rate as previous weight compression methods with negligible accuracy loss. The benefit of CBTD over previous methods is the compatibility with the DN algorithm. Temporal sparsity is achieved through the DeltaLSTM model by applying the delta network algorithm to the LSTM model. The Spartus accelerator is implemented on a Xilinx Zynq 7100 FPGA running at 200 MHz. Evaluated on the TIMIT dataset, Spartus achieves 9.4-TOp/s effective batch-1 throughput and 1.1-TOp/J power efficiency, which is, respectively, 4× and 7× higher than the previous state of the art. Compared to the theoretical peak hardware performance, which runs a dense LSTM layer with 1024 neurons in 46 μs, Spartus runs the same network in 1 μs by jointly exploiting structured spatial sparsity (14× speedup) and temporal sparsity (3.3× speedup) to achieve 46× speedup in total. For lightweight and low-cost edge applications, Edge-Spartus provides over 75-GOp/s effective throughput for arbitrary-sized LSTM-RNNs on a $55$ USB-powered MiniZed board, which will be useful for edge signal processing and mobile robots. Compared to ASIC-based RNN accelerators [17] that mainly focus on low power processing, Spartus achieves significantly higher effective throughput but also massively more power consumption due to its implementation in FPGAs. A future ASIC implementation of Spartus could achieve even higher power efficiency and throughput than the existing FPGA-based systems.

**ACKNOWLEDGMENT**

The authors thank X. Chen and other Sensors Group members for discussion on general design concepts. They also thank A. Rios-Navarro, R. Morales, and A. Linares-Barranco from the University of Seville for creating the baseboard for field-programmable gate arrays (FPGAs).

**REFERENCES**

[1] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural Comput., vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10.1162/neco.1997.9.8.1735.

[2] K. Cho et al., “Learning phrase representations using RNN encoder-decoder for statistical machine translation,” in Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP), 2014, pp. 1724–1734.

[3] D. Amodei et al., “Deep speech 2: End-to-end speech recognition in English and Mandarin,” in Proc. Int. Conf. Mach. Learn., 2016, pp. 173–182.

[4] A. Graves, A.-R. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” in Proc. IEEE Int. Conf. Acoust., Speech Signal Process., May 2013, pp. 6645–6649, doi: 10.1109/ICASSP.2013.6638947.

[5] T. Mikolov, M. Karafiát, L. Burget, J. Černocký, and S. Khudanpur, “Recurrent neural network based language model,” in Proc. 11th Annu. Conf. Int. Speech Commun. Assoc. (INTERSPEECH), Sep. 2010, pp. 1045–1048. [Online]. Available: https://www.itl.vut.cz/research/publication/9362

[6] C. Gao, R. Gehlhar, A. D. Ames, S.-C. Liu, and T. Delbruck, “Recurrent neural network control of a hybrid dynamical transmemoral prosthesi with EdgeDRNN accelerator,” in Proc. IEEE Int. Conf. Robot. Autom. (ICRA), May 2020, pp. 5460–5466.

[7] O. Vinyals et al., “Grandmaster level in starcraft II using multi-agent reinforcement learning,” Nature, vol. 575, no. 7782, pp. 350–354, 2019, doi: 10.1038/s41586-019-1724-z.

[8] S. Mozaffari, O. Y. Al-Jarrah, M. Dianati, P. Jennings, and A. Mozukhitis, “Deep learning-based vehicle behavior prediction for autonomous driving applications: A review,” IEEE Trans. Intell. Transp. Syst., vol. 23, no. 1, pp. 33–47, Jan. 2022.

[9] M. Horowitz, “Computing’s energy problem (and what we can do about it),” in IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers, Feb. 2014, pp. 10–14.

[10] N. P. Jouppi et al., “Ten lessons from three generations shaped Google’s TPUv4: Industrial product,” in Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA), Jun. 2021, pp. 1–14.

[11] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding,” in Proc. Int. Conf. Learn. Represent., 2016, pp. 1–14.

[12] S. Han et al., “ESE: Efficient speech recognition engine with sparse LSTM on FPGA,” in Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays, New York, NY, USA: Association for Computing Machinery, Feb. 2017, pp. 75–84, doi: 10.1145/3020078.3021745.

[13] A. N. Gomez, I. Zhang, K. Sversky, Y. Gal, and G. E. Hinton, “Targeted dropout,” in Proc. CDNARIA Workshop 32nd Conf. Neural Inf. Process. Syst. (NeurIPS), 2018. [Online]. Available: https://openreview.net/pdf?id=HkgbWScuo

[14] W. Wen et al., “Learning intrinsic sparse structures within long short-term memory,” in Proc. Int. Conf. Learn. Represent., 2018, pp. 1–14. [Online]. Available: https://openreview.net/forum?id=HkcfcpRfZJ.

[15] S. Cao et al., “Effective and efficient sparse LSTM on FPGA with bank-balanced sparsity,” in Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays, New York, NY, USA, Feb. 2019, pp. 63–72, doi: 10.1145/3298602.3293898.

[16] M. Wang, Z. Wang, J. Lu, J. Lin, and Z. Wang, “E-LSTM: An efficient hardware architecture for long short-term memory,” IEEE Trans. Emerg. Sel. Topics Circuits Syst., vol. 9, no. 2, pp. 280–291, Jun. 2019, doi: 10.1109/ETCAS.2019.2911739.

[17] D. Kadetotad, S. Yim, V. Berisha, C. Chakrabarti, and J.-S. Seo, “An 8.93 TOPS/W LSTM recurrent neural network accelerator featuring hierarchical coarse-grain sparsity for on-device speech recognition,” IEEE J. Solid-State Circuits, vol. 55, no. 7, pp. 1877–1887, Jul. 2020, doi: 10.1109/JSSC.2020.2992900.

[18] Z. Wang, J. Lin, and Z. Wang, “Accelerating recurrent neural network models: A memory-efficient approach,” IEEE Trans. Very Large Scale Integ. (VLSI) Syst., vol. 25, no. 10, pp. 2763–2775, Oct. 2017, doi: 10.1109/TVLSI.2017.2717950.

[19] S. Wang et al., “C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs,” in Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays, New York, NY, USA, Feb. 2018, pp. 11–20, doi: 10.1145/3174243.3174253.

[20] L. Li et al., “E-RNN: Design optimization for efficient recurrent neural networks in FPGAs,” in Proc. IEEE Int. Symp. High Perfom. Comput. Archit. (HPCA), Feb. 2019, pp. 69–80, doi: 10.1109/HPCA.2019.00028.

[21] D. Neil, J. Lee, T. Delbruck, and S.-C. Liu, “Delta networks for optimized recurrent network computation,” in Proc. 34th Int. Conf. Mach. Learn. (ICML), Sydney, NSW, Australia, Aug. 2017, pp. 2584–2593. [Online]. Available: http://proceedings.mlr.press/v70/neil17a.html

[22] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, “DeltaRNN: A power-efficient recurrent neural network accelerator,” in Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays, New York, NY, USA, Feb. 2018, pp. 21–30, doi: 10.1145/3174243.3174261.

[23] C. Gao, A. Rios-Navarro, X. Chen, S.-C. Liu, and T. Delbruck, “EdgeDRNN: Recurrent neural network accelerator for edge inference,” IEEE J. Emerg. Sel. Topics Circuits Syst., vol. 10, no. 4, pp. 419–432, Dec. 2020, doi: 10.1109/ETCAS.2020.3040300.

[24] A. Aimar et al., “NullHop: A flexible convolutional neural network accelerator based on sparse representations of feature maps,” IEEE Trans. Neural Netw. Learn. Syst., vol. 30, no. 3, pp. 644–656, Mar. 2019, doi: 10.1109/TNNLS.2018.2852335.

[25] Q. Chen et al., “An efficient accelerator for multiple convolutions from the sparsity perspective,” IEEE Trans. Very Large Scale Integ. (VLSI) Syst., vol. 28, no. 6, pp. 1540–1544, Jun. 2020, doi: 10.1109/TVLSI.2020.2976454.

[26] C. Xu et al., “Alternating multi-bit quantization for recurrent neural networks,” in Proc. Int. Conf. Learn. Represent., 2018, pp. 1–13. [Online]. Available: https://openreview.net/forum?id=S19dR9x0b

[27] I. S. Duff, R. G. Grimes, and J. G. Lewis, “Sparse matrix test problems,” ACM Trans. Math. Sofware, vol. 15, no. 1, pp. 1–14, Mar. 1989, doi: 10.1145/62038.62043.
S. Han, C. Gao, S. Braun, I. Kiselev, J. Anumula, T. Delbruck, and S.-C. Liu, “Proving the lottery ticket hypothesis: Finding sparse, trainable neural networks,” in Proc. Int. Conf. Learn. Represent., 2019, pp. 1–42. [Online]. Available: https://openreview.net/forum?id=BJbrFrF7

E. Malach, G. Yehudai, S. Shalev-Schwartz, and O. Shamir, “Proving the lottery ticket hypothesis: Pruning is all you need,” in Proc. 37th Int. Conf. Mach. Learn., vol. 119, A. Singh, Eds., Jul. 2020, pp. 6682–6691. [Online]. Available: https://proceedings.mlr.press/v119/malach20a.html

S. Han et al., “DSD: Dense-sparse-dense training for deep neural networks,” in Proc. Int. Conf. Learn. Represent., 2017, pp. 1–13. [Online]. Available: https://openreview.net/pdf?id=HysOT9xI

C. Gao, S. Braun, I. Kiselev, J. Anumula, T. Delbruck, and S.-C. Liu, “Real-time speech recognition for IoT purpose using a delta recurrent neural network accelerator,” in Proc. IEEE Int. Symp. Circuits Syst. (ISCAS), May 2019, pp. 1–5.

H. Sak, A. Senior, and F. Beaufays, “Long short-term memory recurrent neural network architectures for large scale acoustic modeling,” in Proc. Interspeech, Sep. 2014, pp. 338–342.

M. O’Connor et al., “Fine-grained DRAM: Energy-efficient DRAM for extreme bandwidth systems,” in Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture, Oct. 2017, pp. 41–54. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8686544/

Chang Gao (Member, IEEE) received the B.Eng. degree in electronics from the University of Liverpool, Liverpool, U.K., and Xi’an Jiaotong-Liverpool University, Suzhou, China, in July 2015, the master’s degree in analog and digital integrated circuit design from the Imperial College London, London, U.K., in November 2016, and the Ph.D. degree from the Institute of Neuroinformatics, University of Zurich and ETH Zurich, Zurich, Switzerland, in December 2021.

He will be joining the Delft University of Technology (TU Delft), Delft, the Netherlands, as an Assistant Professor, in 2022. His current research interest includes computer architectures for deep learning with an emphasis on recurrent neural networks.

Tobi Delbruck (Fellow, IEEE) received the B.Sc. degree in physics from the University of California at San Diego, San Diego, CA, USA, in 1986, and the Ph.D. degree from the California Institute of Technology, Pasadena, CA, USA, in 1993.

Since 1998, he has been with the Institute of Neuroinformatics, University of Zürich and ETH Zürich, Zürich, Switzerland, where he is currently a Professor of physics and electrical engineering. His group along with S.-C. Liu focuses on neuromorphic sensory processing, efficient hardware AI, and control for robotics.

Shih-Chii Liu (Fellow, IEEE) received the bachelor’s degree in electrical engineering from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 1983, and the Ph.D. degree in the computation and neural systems program from the California Institute of Technology, Pasadena, CA, USA, in 1997.

She was with various companies, including Gould American Microsystems, San Jose, CA, USA; LSI Logic, Sherman Oaks, CA, USA; and Rockwell International Research Labs, Thousand Oaks, CA, USA. She is currently a Professor and a Group Leader with the Institute of Neuroinformatics, University of Zürich and ETH Zürich, Zürich, Switzerland.