DeltaRNN: A Power-efficient Recurrent Neural Network Accelerator

Chang Gao
Institute of Neuroinformatics,
University of Zurich and ETH Zurich
Zurich, Switzerland
chang@ini.uzh.ch

Daniel Neil*
Institute of Neuroinformatics,
University of Zurich and ETH Zurich
Zurich, Switzerland
daniel.l.neil@gmail.com

Enea Ceolini
Institute of Neuroinformatics,
University of Zurich and ETH Zurich
Zurich, Switzerland
eceoli@ini.uzh.ch

Shih-Chii Liu
Institute of Neuroinformatics,
University of Zurich and ETH Zurich
Zurich, Switzerland
shih@ini.ethz.ch

Tobi Delbruck
Institute of Neuroinformatics,
University of Zurich and ETH Zurich
Zurich, Switzerland
tobi@ini.uzh.ch

ABSTRACT

Recurrent Neural Networks (RNNs) are widely used in speech recognition and natural language processing applications because of their capability to process temporal sequences. Because RNNs are fully connected, they require a large number of weight memory accesses, leading to high power consumption. Recent theory has shown that an RNN delta network update approach can reduce memory access and computes with negligible accuracy loss. This paper describes the implementation of this theoretical approach in a hardware accelerator called "DeltaRNN" (DRNN). The DRNN updates the output of a neuron only when the neuron's activation changes by more than a delta threshold. It was implemented on a Xilinx Zynq-7100 FPGA. FPGA measurement results from a single-layer RNN of 256 Gated Recurrent Unit (GRU) neurons show that the DRNN achieves 1.2 TOp/s effective throughput and 164 GOp/s/W power efficiency. The delta update leads to a 5.7x speedup compared to a conventional RNN update because of the sparsity created by the DN algorithm and the zero-skipping ability of DRNN.

ACM Reference Format:
Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbruck. 2018. DeltaRNN: A Power-efficient Recurrent Neural Network Accelerator. In *FPGA '18: 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 25–27, 2018, Monterey, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3174243.3174261

1 INTRODUCTION

Recurrent Neural Networks (**RNNs**) are fully-connected single- or multi-layered networks with complex neurons that have multiple memory states and enabled state-of-art accuracies in tasks involving temporal sequences [27] such as automatic speech recognition [2, 9] and natural language processing [23]. The prediction accuracy of RNNs is further improved by adding gating units to control the data flow in and between neurons. In deep learning, Long Short-Term Memory (**LSTM**) [16] and Gated Recurrent Unit (**GRU**) [6] are two major neuron models used in gated RNNs. The gating units in the LSTM and GRU models help to mitigate the well-known vanishing gradient problem encountered during the training process.

A challenge in deploying RNN applications in mobile or always-on applications is access to hardware that achieves high power efficiency. For mobile applications, edge inference is preferable because of lower latency, reduced network bandwidth requirement, robustness to network failure, and better privacy. Recent neural network applications use GPUs ranging from high-end models such as the NVIDIA Pascal Titan X GPU to embedded system on chip (**SoC**) such as the Kepler GPU in Tegra K1 SoC or smartphone processor embedded GPU such as the Samsung Exynos Mali. The server- or desktop-targeted Titan X consumes about 200 W and achieves a power efficiency around 5 GOp/s/W in inference of an LSTM layer with 1024 neurons [12]. Embedded CPUs and GPUs consume about 1 W, but their RNN power efficiency is actually much lower (around 8 MOp/s/W during inference with an LSTM network with 2 layers of 128 LSTM neurons each, as measured in [5]). For mobile GPUs, the low efficiency is from poor matching of memory architecture to GPU, especially for small RNNs.

The energy figures from Horowitz [17] for a 45nm process show that Dynamic Random Access Memory (**DRAM**) access consumes several hundred times more energy than arithmetic operations. The power consumption of the LSTM RNN in a complete mobile speech recognition engine [22] can be estimated. The RNN is a 5-layer network with 500 units per layer, that is updated at 100 Hz. The weight matrices are too large to be stored in economical SRAM and so must be fetched from off-chip DRAM. Using the Horowitz numbers results in a power consumption of about 0.2 W with most consumed by DRAM access. Thus RNN inference energy is dominated by memory access cost; however, achieving high throughput requires high memory bandwidth for weight fetching in order to keep arithmetic units loaded as fully as possible. The key to improving power efficiency (number of arithmetic operations/W) is to reduce the total memory access therefore decreasing the energy consumption, while keeping arithmetic units fully utilized to maintain high throughput.

*Currently at BenevolentAI.

FPGA '18, February 25–27, 2018, Monterey, CA, USA

ACM ISBN 978-1-4503-5614-5/18/02.
https://doi.org/10.1145/3174243.3174261

Sparsity in network activations or input data is a property that can be used to achieve high power efficiency. In a Matrix-Vector Multiplication (**MxV**) between a neuron activation vector and a weight matrix, zero elements in the vector result in zero partial sums that do not contribute to the final result. The multiplications between zero vector elements and their corresponding weight columns can be skipped to save memory access of weight columns. Moreover, it is also possible to skip any multiplication between a zero weight element and a non-zero vector element to further reduce operations and memory access, though this feature has not been adopted in this work yet. A common way to create sparsity in neural network parameters is weight compression, which is shown in works [14, 19]. The Delta Network algorithm [25] creates sparsity in input and activation vectors by exploiting their temporal dependency.

Although GPUs offer high peak throughput, they may suffer from irregular execution paths and memory access patterns in RNN inference, which is caused by recurrent connections and limited data reuse of RNNs [3]. In this case, it is useful to consider hardware architectures specialized for RNN inference that enhance parallelism by matching memory bandwidth to available arithmetic units. RNN inference accelerators based on FPGA have been explored because of the potentially higher power efficiency compared with GPUs. Previous works include LSTM accelerators [4, 5, 11, 21], a GRU accelerator [26] and a Deep Neural Network (**DNN**) accelerator [10] that is able to run LSTM inference. These implementations do not capitalize on the data sparsity of RNNs. The Efficient Speech Recognition Engine (ESE) proposed by Han et al. [13] employs a load-balance-aware pruning scheme, including both pruning [15] and quantization. This pruning scheme compresses the LSTM model size by 20x and a scheduler that parallelizes operations on sparse models. They achieved 6.2x speedup over the dense model by pruning the LSTM model to 10% non-zeros. However, none of these works use the temporal property of RNN data.

In this paper, a GRU-RNN accelerator architecture called the DeltaRNN (**DRNN**) is proposed. This implementation is based on the Delta Network (**DN**) algorithm that skips dispensable computations during network inference by exploiting the temporal dependency in RNN inputs and activations [25] (Sec. 2.2). The system was implemented on an Xilinx Zynq-7100 FPGA controlled by a dual ARM Cortex-A9 CPU. To provide sufficient bandwidth for arithmetic units and less external memory access, DRNN V1.0 stores the network weight matrix in BRAM blocks. It was tested on the TIDIGITS dataset using an RNN model with one GRU layer of 256 neurons.

2 BACKGROUND

2.1 Gated Recurrent Unit

A GRU RNN has similar prediction accuracy to an LSTM RNN but lower complexity of computation. We implemented the GRU which requires a smaller number of network parameters (therefore less hardware resource) than the LSTM. Fig. 1 shows the data flow within a GRU neuron.

The GRU neuron model has two gates–a reset gate r and an update gate u–and a candidate hidden state c. The reset gate determines the amount of information from the previous hidden state

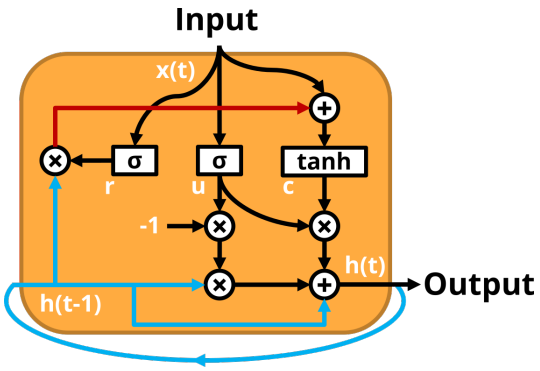

Figure 1: Data flow of GRU

that will be added to the candidate hidden state. The update gate decides to what extent the activation h should be updated by the candidate hidden state to enable a long-term memory. The GRU formulation used in this paper is shown below:

$$r(t) = \sigma[W_{xr}x(t) + W_{hr}h(t-1) + b_r] \quad (1)$$

$$u(t) = \sigma[W_{xu}x(t) + W_{hu}h(t-1) + b_u] \quad (2)$$

$$c(t) = \tanh[W_{xc}x(t) + r(t) \odot (W_{hc}h(t-1)) + b_c] \quad (3)$$

$$h(t) = (1 - u(t)) \odot h(t-1) + u(t) \odot c(t) \quad (4)$$

where x is the input vector, h the activation vector, W the weight matrix, b the bias and r, u, c correspond to the reset gate, update gate and candidate activation respectively. σ and $\odot$ signify logistic sigmoid function and element-wise multiplication respectively.

2.2 Delta Network Algorithm

This section explains the principle of the DN algorithm [25] and show how a conventional GRU-RNN model with full updates can be converted to a delta network.

The DN algorithm reduces both memory access and arithmetic operations by exploiting the temporal stability of RNN inputs and outputs. Computations associated with a neuron activation that has a small amount of change from its previous timestep can be skipped. As shown in Fig. 2, all gray circles represent neurons whose corresponding computations are skipped. The previous research on the DN algorithm [25] demonstrated for the TIDIGITS audio digit recognition benchmark that the algorithm can achieve 8x speedup with 97.5% accuracy when the network was trained as a delta network without considering sparsity in the weight matrix. Pre-trained networks can also be greatly accelerated as delta networks. The large Wall Street Journal (**WSJ**) speech recognition benchmark showed speedup of 5.7x with word error rate of 10.2%, which is the same with using a conventional RNN [25].

Fig. 3 shows how skipping a single neuron saves multiplications of an entire column in all related weight matrices as well as fetches of the corresponding weight elements. The following equations describe the conversion between an MxV with full update and one with delta updates:

$$y(t) = Wx(t) \quad (5)$$

$$y(t) = W\Delta x(t) + y(t-1) \quad (6)$$

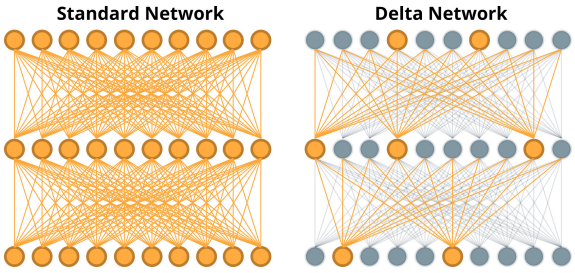

Figure 2: Comparison between a standard gated RNN network (left) and a sparse delta network (right)

where $\Delta x\,(t) = x\,(t) - x\,(t-1)$ and $y\,(t-1)$ is the MxV result from the previous timestep. The MxV in equation (6) becomes a sparse MxV if all computations with respect to small $\Delta x(t)$ elements are ignored. As shown in Fig. 3, the sparser the $\Delta x(t)$ vector, the more memory access and arithmetic operations are saved.

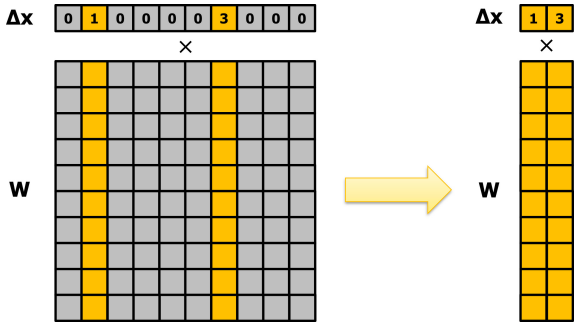

Figure 3: Skipping neuron updates save multiplications between input vectors and columns that correspond to zero $\Delta x(t)$ (also the behavior of Matrix-Vector Multiplication Channel discussed in Section 3.2.2)

Assuming the length of all vectors is n and dimension of the weight matrix W is $n \times n$, the computation cost for calculating a dense MxV is n^2 while the computation cost for calculating a DN MxV is $o_c \cdot n^2 + 2n$, where o_c is the occupancy[1] of the delta vector $\Delta x(t)$. The term $2n$ exists because calculating the delta vector $\Delta x(t)$ and adding $y(t-1)$ to $W\Delta x(t)$ respectively needs n operations. As for memory access, to calculate a dense MxV, $o_c \cdot n^2$ weight elements and n vector elements has to be fetched. The DN MxV needs to fetch $o_c \cdot n^2$ weight elements, $2n$ vector elements for $\Delta x(t)$, n vector elements for $y(t-1)$ and finally write n vector elements for $y(t)$. Thus, the theoretical computation speedup and memory access reduction approaches $1/o_c$ when $n \to \infty$. A summary of the computation cost and memory cost is shown by:

$$C_{\text{comp, dense}} = n^2 \tag{7}$$

$$C_{\text{comp, sparse}} = o_c \cdot n^2 + 2n \tag{8}$$

$$C_{\text{mem, dense}} = n^2 + n \tag{9}$$

[1]Occupancy is defined as the ratio of non-zero elements to all elements of a vector or a matrix

$$C_{\text{mem, sparse}} = o_c \cdot n^2 + 4n \tag{10}$$

$$\text{Speedup} = \frac{C_{\text{comp, dense}}}{C_{\text{comp, sparse}}} \approx \frac{1}{o_c} \tag{11}$$

$$\text{Memory Access Reduction} = \frac{C_{\text{mem, dense}}}{C_{\text{mem, sparse}}} \approx \frac{1}{o_c} \tag{12}$$

To skip the computations related to any small $\Delta x\,(t)$, the delta threshold Θ is introduced to decide when a delta vector element can be ignored. The change of a neuron's activation is only memorized when it is larger than Θ. Furthermore, to prevent the accumulation of error with time, only the last activation value that has a change larger than the delta threshold is memorized. This is defined by the following equation sets:

$$\hat{x}(t-1) = \begin{cases} x(t-1) & ,|x(t) - \hat{x}(t-1)| > \Theta \\ \hat{x}(t-2) & ,|x(t) - \hat{x}(t-1)| \le \Theta \end{cases} \tag{13}$$

$$\hat{h}(t-2) = \begin{cases} h(t-2) & ,\left|h(t-1) - \hat{h}(t-2)\right| > \Theta \\ \hat{h}(t-3) & ,\left|h(t-1) - \hat{h}(t-2)\right| \le \Theta \end{cases} \tag{14}$$

$$\Delta x\,(t) = \begin{cases} x\,(t) - \hat{x}\,(t-1) & ,|x\,(t) - \hat{x}\,(t-1)| > \Theta \\ 0 & ,|x\,(t) - \hat{x}\,(t-1)| \le \Theta \end{cases} \tag{15}$$

$$\Delta h\,(t-1) = \begin{cases} h\,(t-1) - \hat{h}\,(t-2) & ,\left|h\,(t-1) - \hat{h}\,(t-2)\right| > \Theta \\ 0 & ,\left|h\,(t-1) - \hat{h}\,(t-2)\right| \le \Theta \end{cases} \tag{16}$$

where memorized changes $\Delta x(t)$ and $\Delta h(t-1)$ are calculated by using $\hat{x}(t-1)$ and $\hat{h}(t-2)$. Next, using (13), (14), (15) and (16), the conventional GRU equation set can be transformed into its delta network version:

$$M_r(t) = W_{xr}\Delta x(t) + W_{hr}\Delta h(t-1) + M_r(t-1) \tag{17}$$

$$M_u(t) = W_{xu}\Delta x(t) + W_{hu}\Delta h(t-1) + M_u(t-1) \tag{18}$$

$$M_{cx}(t) = W_{xc}\Delta x(t) + M_{cx}(t-1) \tag{19}$$

$$M_{ch}(t) = W_{hc}\Delta h(t-1) + M_{ch}(t-1) \tag{20}$$

$$r(t) = \sigma[M_r(t)] \tag{21}$$

$$u(t) = \sigma[M_u(t)] \tag{22}$$

$$c(t) = \tanh[M_{cx}(t) + r(t) \odot M_{ch}(t)] \tag{23}$$

$$h(t) = [1 - u(t)] \odot h(t-1) + u(t) \odot c(t) \tag{24}$$

where $M_r(0) = b_r$, $M_u(0) = b_u$, $M_{cx}(0) = b_c$, $M_{ch}(0) = 0$.

3 IMPLEMENTATION

The DN algorithm can theoretically reduce arithmetic operations and memory access by reducing weight fetches. The main target of the DRNN accelerator is to realize efficient zero-skipping on sparse and irregular data patterns of $\Delta x(t)$ and $\Delta h(t-1)$.

Fig. 4 shows the overview of the whole system. The Programmable Logic (**PL**) part is implemented on a Xilinx Zynq-7100 FPGA chip running at 125 MHz. An AXI Direct Memory Access (**DMA**) module converts between AXI4-Stream (**AXIS**) and full AXI4 so that data can be transferred between the PL and the Processing System (**PS**). The PS is the Dual ARM-Cortex A9 CPU on the Zynq SoC. The data transfer between DRNN and AXI DMA is managed in packets. The Memory Mapped to Stream (**MM2S**) interrupt and the Stream to Memory Mapped (**S2MM**) interrupt are respectively used to indicate the end of corresponding data transfers. Read and

write operations on DDR3 memory are managed by the DRAM controller of the PS [28].

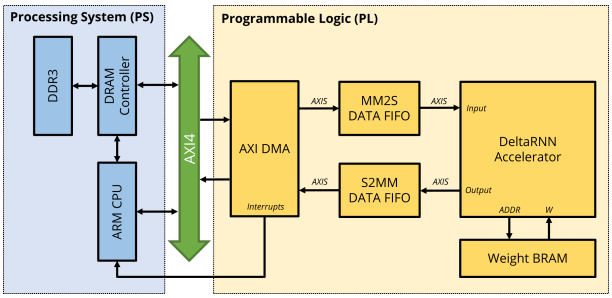

Figure 4: Acceleration system overview

The input and output interface of the DRNN accelerator uses the AXIS protocol to stream data in from the MM2S Data FIFO and out to the S2MM Data FIFO. Both the input and output interfaces are 64-bit wide in order to transfer 4 16-bit values per clock cycle. The weight BRAM block consumes 400 36-Kbit BRAM blocks (~ 1.76 MB) to store all weight matrices and biases on-chip to provide sufficient bandwidth. Input vectors are stored in DRAM to be transferred to DRNN during runtime and output vectors are written back to DRAM immediately after being produced by DRNN. Previous research shows that RNNs with quantized fixed-point parameters down to 8 and even 4 bits can still work well [18, 20]. In this work, to demonstrate the benefit of the DRNN architecture on performance gain with minimum RNN accuracy loss on practical applications, we quantize all 32-bit floating-point parameters used by the DRNN including GRU input/output vectors, weights and biases into fixed-point 16-bit Q8.8 integers by **Fixed16** = round $(256 \times$ **Float32**$)$.

3.1 DRNN Architecture

The top-level block diagram of the DRNN Accelerator is shown in Fig. 5. It is composed of three main modules, the Input Encoding Unit (**IEU**), the **MxV** Unit which is controlled by the MxV controller, and the Activation Pipeline (**AP**).

The function of the IEU is to execute subtractions and comparisons between the input vectors from the current and previous timesteps to generate delta vectors. The MxV Unit executes Multiply-Accumulate (**MAC**) operations using the sparse IEU output. It contains 768 MAC units that performs 16-bit multiplications and 32-bit accumulations on signed integers. Accumulation results from the MxV Unit are sent to the AP module which computes the activation of the GRU layer for the current timestep. The architecture and function of each block is next presented in detail.

3.1.1 Input Encoding Unit. The IEU encodes dense input vectors $x(t)$ and $h(t-1)$ into sparse delta input vectors $\Delta x(t)$ and $\Delta h(t-1)$; however, the sparsity of these delta input vectors is not predictable and can only be known at runtime, leading to irregular data patterns. Hence another function of IEU is to format non-zero elements of delta input vectors so that they can be fetched one by one in consecutive clock cycles. This IEU is the most significant and complicated module of the DRNN.

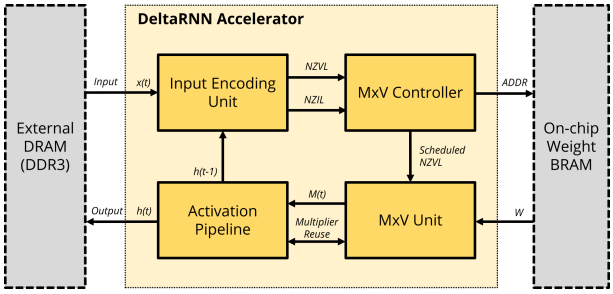

Figure 5: Top-level block diagram of the DRNN Accelerator

The structure of the IEU is shown in Fig. 6. The IEU has two identical parts that are responsible for generating $\Delta x(t)$ and $\Delta h(t-1)$ respectively. The width of the inputs of the two parts are 64 bits and 512 bits. Since both $x(t)$ and $h(t-1)$ elements are 16-bit Q8.8 integers, the two IEU parts respectively consume 4 elements of $x(t)$ and 32 elements of $h(t-1)$ per clock cycle, both of which should be set as large as possible to enhance performance but are respectively limited by the AXI-DMA bandwidth and timing requirements. The same number of delta vector elements are then calculated by the Delta Encoder by subtracting the current input from the input of last timestep stored in the Previous Time (**PT**) register file and comparing the result with the delta threshold to decide if the magnitude of change should be dropped or saved into the register file in the Delta Scheduler. After being processed by the Delta Encoder, $\hat{x}(t-1)$, $\hat{h}(t-2)$ are selected from $x(t)$, $h(t-1)$ respectively and then written into the PT Register to be used to calculate delta vectors for the next timestep. The Delta Scheduler in either of the two parts of IEU can generate 2 non-zero $\Delta x(t)$ elements or 2 non-zero $\Delta h(t-1)$ elements. Next, any non-zero values in delta vectors and their corresponding indices are respectively allocated in two groups of FIFOs to form the Non-Zero Value List (**NZVL**) and the Non-Zero Index List (**NZIL**), which is a variant of the sparsity map used in [1].

The latency of calculating a $\Delta h(t-1)$ vector with 1024 elements is at best $1024/32 = 32$ cycles with all elements to be zeros (0% occupancy) and at worst $1024/2 = 512$ cycles with all elements to be non-zero (100% occupancy). The DRNN is designed to calculate MxV column-wise so that the MxV computation can be started immediately after a valid delta vector element is generated. In this way, multiplications between a matrix column and a delta vector element can be easily parallelized because of the locality of the non-zero delta vector element; otherwise, if MxV is calculated row-wise, the delta encoding process might introduce a huge overhead when running a large model.

Moreover, since the IEU part for generating $\Delta x(t)$ consumes 4 elements per cycle, any input vector with an odd number of elements has to be extended to a length that is a multiple of 4 using zero-padding (this does not apply to $h(t-1)$ vectors because only even numbers of hidden layer neurons are supported by DRNN). For example, if the length of the input vectors $x(t)$ is 39, then 1 extra zero will be appended to the end of $x(t)$. Thus it takes at best 10 clock cycles and at worst 20 cycles for IEU to generate a $\Delta x(t)$ vector with 40 elements.

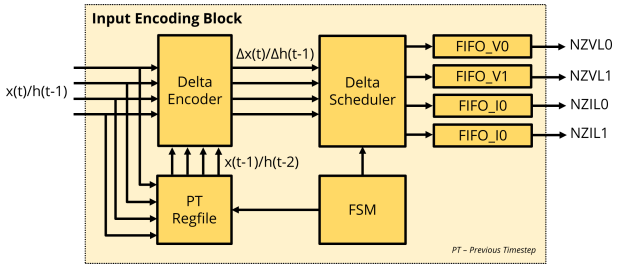

Figure 6: Block diagram of the IEU

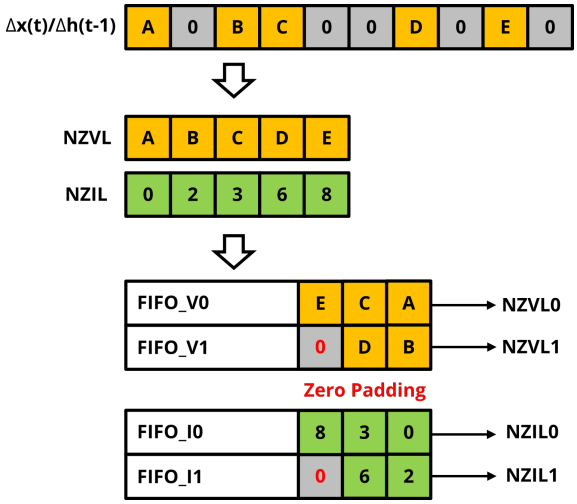

Figure 7: Principle of generating and allocating NZVL and NZIL

As shown in Fig. 7, a complete NZVL consists of all non-zero elements of $\Delta x(t)$ or $\Delta h(t-1)$ and the corresponding NZIL contains indices of elements in the NZVL and can be encoded into addresses for fetching corresponding weight columns. Either NZVL or NZIL is split and allocated in two FIFOs in ascending order of indices. Zero padding is conducted at the end of the allocation if the number of non-zero values is an odd number.

*3.1.2 Matrix-Vector Multiplication (**MxV**) Unit.* The MxV Unit performs dense matrix and sparse vector multiplications to skip columns of computations that correspond to zero vector elements in $\Delta x(t)$ and $\Delta h(t-1)$, which is different from previous works on accelerating sparse matrix-vector multiplications that mainly exploit sparsity in matrices [7, 8, 29]. The MxV Unit is composed of 3 channels, R, U and C. Channels R and U are respectively responsible for calculating memories $M_r(t)$ and $M_u(t)$ while Channel C calculates both $M_{cx}(t)$ and $M_{ch}(t)$. Each channel has 128 clusters of multipliers (**MUL**) and 128 clusters of summation adders (**ADD**) both with two instances per cluster. This is equivalent to 256 MAC units per channel. Multipliers are instantiated using DSP blocks to perform multiplications on 16-bit signed integers. Summation adders are synthesized by Look-up Tables (**LUTs**) and perform 32-bit summation. Partial sums are accumulated on the accumulation registers (**ACC REG**). The data flow of an MxV channel is shown in Fig. 8.

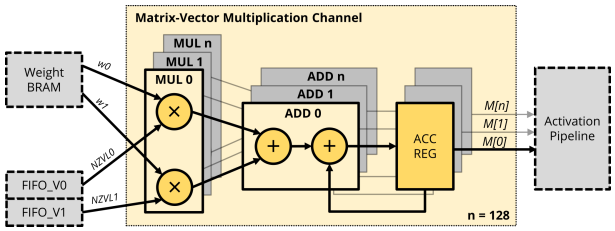

Figure 8: MxV channel

As shown in Fig. 8, in each multiplier cluster (**MUL**), the two operands of one of the multipliers are NZVL0 and the corresponding weight element in the column addressed by NZIL0. Operands of the other multipliers are driven by NZVL1 and the corresponding weight element in another column addressed by NZIL1. Operands w0 and w1 are provided by RAMB18E1 cells in the Weight BRAM block with each configured in true dual-port mode. The MxV channel is fully pipelined to fetch operands in every following clock cycle once launched. When NZVL and NZIL FIFOs are not empty, the channel starts to fetch data in all FIFOs simultaneously including any padded zeros to ensure that NZVL and NZIL are synchronized. Since $\Delta h(t-1)$ is generally longer than $\Delta x(t)$, all MxV channels calculate $W_x \Delta x(t)$ before $W_h \Delta h(t-1)$ to hide the overhead of generating $\Delta h(t-1)$ under the computation time. Accumulation results, $M_r(t)$, $M_u(t)$, $M_{cx}(t)$ and $M_{ch}(t)$, are held in ACC REGs. Channel C has two clusters of ACC REGs to store $M_{cx}(t)$ and $M_{ch}(t)$ respectively while Channels R and U each have one cluster of ACC REGs to store $M_r(t)$ and $M_u(t)$ respectively.

Fig. 9 illustrates the computation pattern of MxV channels. One MxV channel can simultaneously calculate multiplications between two non-zero delta vector elements and weight elements in two corresponding columns, reducing the total number of fetching/writing accumulation registers by approximately 2x. The MxV channel calculates the MxV in a column-wise style. The Channel Width equals to the number of multiplier clusters in each MxV channel, so that it denotes the amount of column elements that can be processed by the channel in each clock cycle. The Channel Worksize indicates how many clock cycles the MxV channel needs to finish processing a column. In this design, the Channel Width is 128 and the Channel Worksize is 2. It takes 2 clock cycles for each MxV channel to process 2 columns.

*3.1.3 Activation Pipeline (**AP**).* The AP produces the final hidden layer activation from the accumulation results held in ACC REGs. Fig. 10 shows the data flow of the AP, which has 6 pipeline stages, S0-S5. The GRU formulation is divided into 6 steps correspond to each pipeline stage.

The latency of the AP affects the utilization rate of the MxV Unit. Although the part of IEU that processes input $x(t)$ keeps working whenever there are new input vectors coming from the M2SS Data FIFO, the other part that works on $h(t-1)$ must be stalled until a new activation vector is generated by the AP. The condition to launch the MxV Unit is when both NZVL FIFOs for $\Delta x(t)$ and $\Delta h(t-1)$ are not empty. Thus, during this period, the MxV Unit

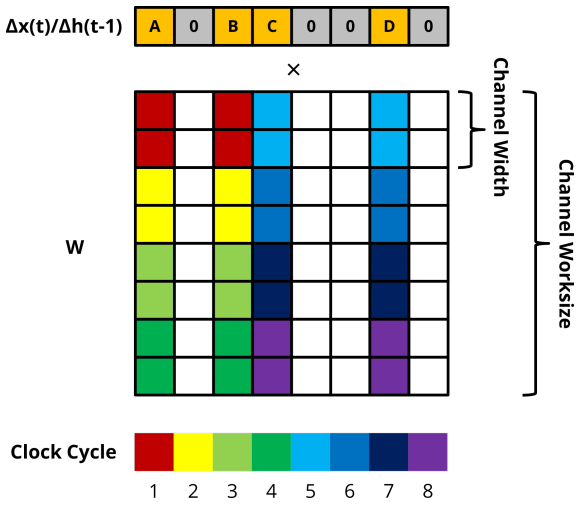

Figure 9: MxV Channel computation pattern

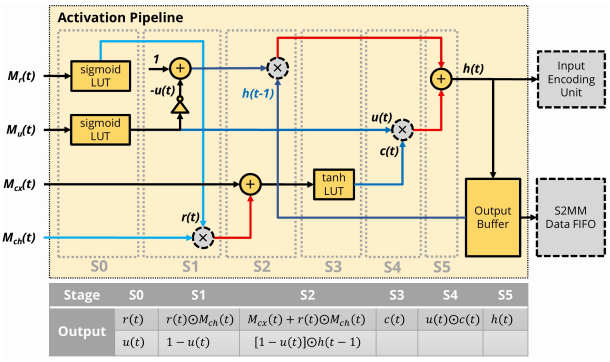

Figure 10: Data flow of the Activation Pipeline module; blue arrows denote data paths from AP to MxV multipliers and red arrows denote those from MxV multipliers to AP

is in idle state due to no new non-zero $\Delta h(t-1)$ elements coming from the IEU. When running larger models, the MxV Unit will run for more clock cycles per timestep compared to the AP and thus the MxV Unit utilization rate is improved. In this case, to enhance the utilization rate of the MxV Unit, multiplications in AP are conducted by reusing the MxV Unit. Multiplexers and demultiplexers are used to control the data paths between AP and multipliers in MxV Unit. Adders in the AP are synthesized by LUT. Adders in S1 and S5 perform 16-bit integer summation and those in S2 perform 32-bit integer summation to calculate $M_c(t) = M_{cx}(t) + M_{ch}(t)$. 32-bit Q16.16 outputs of 'virtual' multipliers in S2 and S4 are transformed into 16-bit Q8.8 before reaching adders in S5. Sigmoid and tanh functions in the AP are realized by using the Range Addressable Lookup Table (RALUT) [24]. A RALUT can save hardware resources by quantizing the non-linear functions within a given input range and any input that exceeds the range will give an output which is saturated to the maximum or minimum of the corresponding non-linear function. The precision and range of the inputs and outputs of sigmoid and tanh RALUTs are shown in Table 1.

Table 1: Precision and range of inputs and outputs of RALUTs in AP

	sigmoid	tanh
Input Precision	signed 16-bit Q8.8	signed 16-bit Q8.8
Sampling Range	[-0x0008,0x0008)	[-0x0008,0x0008)
Sampling Points	4096	4096
Output Precision	unsigned 9-bit Q1.8	signed 10-bit Q2.8
Output Range	[0x0000,0x0100]	[-0xFF00, 0x0100]

The activation vector is stored into the Output Buffer, which can independently write outputs into the S2MM Data FIFO so that computations for the next timestep can be immediately started after the activation vector of current timestep is generated.

4 RESULTS

4.1 Experimental Setup

Vivado 2017.2 was used for synthesis and implementation of the design. After place and route, the system is able to operate at 125 MHz. As shown in Fig. 11, the system is implemented on an Xilinx Zynq-7000 All Programmable SoC Mini-Module Plus (MMP) system-on-a-module (SOM) mounted on a customized baseboard with a power module. It has a dual ARM Cortex-A9 CPU, a Kintex-7 XC7Z100 FPGA, and 1 GB DDR3 SDRAM.

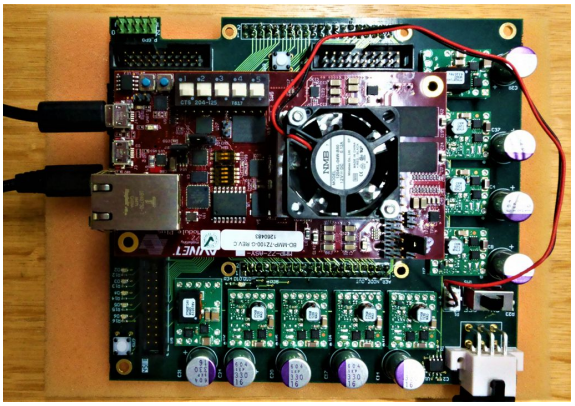

Figure 11: Our customized baseboard supporting Xilinx Zynq-7100 All Programmable SoC Mini-Module Plus from AVNET

DRNN is used to accelerate a single-layer GRU-RNN with 256 neurons that forms part of a classifier to enable real-time spoken digit recognition. The network structure is shown in Fig. 12. The GRU RNN is trained by applying the DN algorithm with different delta thresholds Θ from 0x00 to 0x80 [2] on the TIDIGITS dataset using 32-bit single precision floating-point parameters. The test set has 128 samples, each of which has different numbers of timesteps due to the different audio sequence lengths. Outputs of the GRU RNN layer are processed by two Fully-Connected (**FC**) layers to

[2]The delta threshold is in 16-bit Q8.8 format. For example, 0x80 in Q8.8 corresponds to 0.5 in 32-bit floating-point numbers.

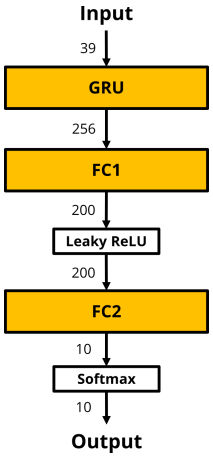

Figure 12: Network structure of the test RNN model

generate the final classification results. Computations of FC layers are handled by the ARM CPU. For training, all samples were zero-padded to have the same length of 249 timesteps. The final classification is done for all the samples on their last timestep. The input dimension is 39 and the input vectors are quantized into 16-bit Q8.8 fixed-point numbers. They are stored in the off-chip DRAM and are transferred to the DRNN one sample at a time during runtime using the AXI-DMA. Hidden layer outputs are written back to the off-chip DRAM to be used by the CPU to produce classification results.

The dimensions of the weight matrices corresponding to inputs and activations are 256×39 and 256×256 respectively. All matrices are quantized into the same format as the input vectors, which gives the total size of 0.43 MB, which are initialized in BRAM blocks together with biases. The test bench is a bare metal C program compiled in the Xilinx SDK environment and is passed to Zynq MMP by USB-JTAG on the baseboard. The range of parameter values before and after quantization is summarized in Table 2.

Table 2: Quantization of the input vector and weight matrices (trained at $\Theta = 0.5$)

	Range (Float-32)	Range (Fixed-16 Q8.8)
$x(t)$	[-8.3845, 12.1637]	[0xF79E, 0x0C2A]
W_{xr}	[-0.4260, 0.3914]	[0xFF93, 0x0064]
W_{xu}	[-0.4505, 0.4107]	[0xFF8D, 0x0069]
W_{xc}	[-0.3862, 0.3649]	[0xFF9D, 0x005D]
W_{hr}	[-0.4249, 0.4426]	[0xFF93, 0x0071]
W_{hu}	[-0.5975, 0.4878]	[0xFF67, 0x007D]
W_{hc}	[-0.4617, 0.4566]	[0xFF8A, 0x0075]

4.2 Hardware Resource Utilization

The hardware resource utilization percentage is shown in Table 3. The maximum number of DSPs used in this design is limited by the maximum bandwidth that RAMB18E1 cells in the weight BRAM block can provide to enable a weight fetch in 1 clock cycle for each DSP cell and also the available LUTs to synthesize adders in the MxV Unit.

Table 3: Hardware utilization of DRNN

	FF	LUT	DSP	BRAM
Available	554800	277400	2020	755
Used	119260	261357	768	457.5
Percentage	21.50%	94.22%	38.02%	60.60%

4.3 Performance

4.3.1 Numerical Accuracy. RNN models trained at different delta thresholds are first computed by an Intel i7-8700k CPU in 32-bit floating-point precision on the 128 test samples. Then the numerical accuracy is calculated as the percentage of same classification results generated by DRNN in 16-bit fixed-point precision compared to those by the CPU. Results shown in Fig. 13 indicate that the same delta threshold Θ should be used in both training and inference to achieve better numerical accuracy. For example, the model with the best numerical accuracy 98.43% (126 out of 128 correct classifications vs. CPU) achieved during inference at $\Theta = 0x80$ is trained also at $\Theta = 0x80$. There is no numerical accuracy loss when $\Theta <= 0x40$ if using the same Θ in both training and inference.

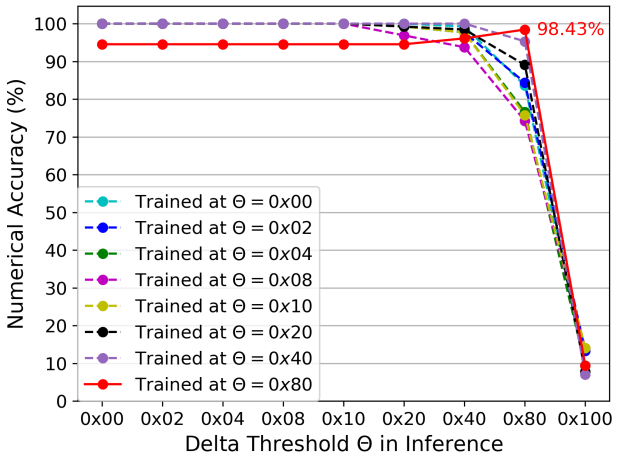

Figure 13: Numerical accuracy of classification results obtained by DRNN running test RNN models trained at different delta thresholds

4.3.2 Throughput. According to the standard GRU formulation, the total Operations per Timestep (OPsT) is:

$$\text{OPsT} = 6 \times \text{LX} \times \text{LH} + 6 \times \text{LH} \times \text{LH} = 453120 \quad (25)$$

where LX is the length of the input vectors and LH is the length of the activations. Since MxV operations dominate the total number of operations, all element-wise multiplications, additions and nonlinear functions are ignored. For this test model, LX = 39 and LH = 256. Then the effective throughput is defined as:

$$\text{Eff. Throughput} = \frac{\text{OPsT} \times \text{Timesteps}}{\text{Time}} \quad (26)$$

The effective throughput of the DRNN is evaluated by streaming 128 input samples to DRNN and measuring the time to process all samples. Padded zeros in each sample are removed. The total number of timesteps of the 128 samples is 12263 leading to a total number of operations equaling $12263 \times 453120 = 5.56$ GOp. Time used by DRNN to finish computing all GRU layer outputs and corresponding effective throughputs at each delta threshold are shown in Table 4. Increasing Θ increases throughput. The highest throughput of 1.2 TOp/s is obtained at Θ = 0x80, which is the delta threshold where optimal throughput with negligible numerical accuracy loss can be achieved. According to Fig. 13 and our previous research [25], delta thresholds larger than 0.5 may cause dramatic accuracy loss.

Table 4: DRNN effective throughput with respect to delta threshold

Delta Threshold	Time [ms]	Eff. Throughput [GOp/s]
0x00	26.43	210.37
0x02	22.77	244.18
0x04	18.89	294.34
0x08	14.61	380.56
0x10	11.12	500.00
0x20	8.14	683.05
0x40	6.04	920.53
0x80	4.64	1198.28

Fig. 14 shows the occupancy of delta input vectors Δx and delta activation vectors Δh averaged across all timesteps, as well as the speedup of GRU layer computation with respect to the delta threshold Θ. When Θ = 0x00 the occupancy of either Δx or Δh is not 100%, indicating that both inputs and activations already have some sparsity without the delta threshold. When Θ = 0x80, the occupancy of Δx and Δh are respectively reduced to 57% and 2%, which means that the total arithmetic operations and memory access are reduced by 43% and 98% for Δx and Δh, respectively. Thanks to the zero-skipping capability, DRNN achieved 5.7x speedup when Θ = 0x80 compared to that when Θ = 0x00. Together with Fig. 13, Fig. 14 also indicates that DRNN can achieve 4.4x speedup without any numerical accuracy loss on the test samples.

The occupancy of Δx is much higher than that of Δh at $\Theta = 0x80$, which indicates that the possibility of IEU to give an invalid output is much higher for Δh than for Δx. For example, if the occupancy of Δh is zero at any timestep, it will take 8 clocks for IEU to consume a Δh vector with 256 elements without generating any valid operand that is useful for the MxV Unit. To reduce the possibility of the MxV Unit idling at the beginning of IEU encoding delta vectors, DRNN calculates $W_h \Delta h(t-1)$ after $W_x \Delta x(t)$ because Δx might provide valid operands more frequently.

4.3.3 MAC Computation Efficiency. The MAC computation efficiency measures how efficiently the hardware makes use of MAC units and is defined as:

$$\text{MAC Comp. Efficiency} = \frac{\text{Eff. Throughput}}{\text{Peak Throughput}} \tag{27}$$

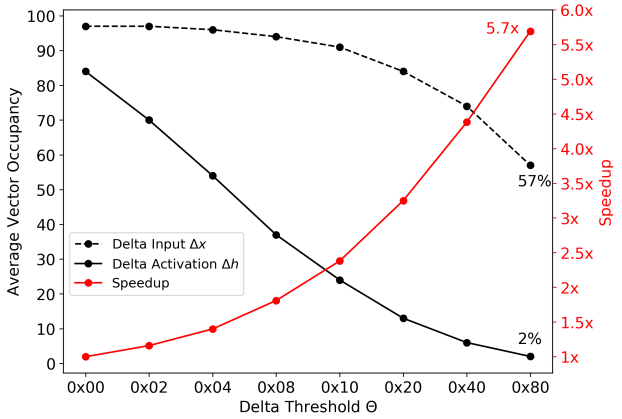

Figure 14: Average vector sparsity and speedup with respect to delta threshold

where peak throughput is defined as $2 \times \#\text{MAC} \times f$ and #MAC is the total number of MAC units, each of which contains a multiplier and an accumulator. Thus the peak throughput of DRNN is calculated as $2 \times 768 \times 125 \times 10^6 = 192$ GOp/s and the MAC computation efficiency of DRNN is $1198/192 = 623.96\%$ for this example workload and delta threshold.

4.3.4 DSP Utilization Efficiency. Here we propose a method to estimate the efficiency of utilizing DSP blocks for MAC computations without considering the bandwidth limitation of memory access. For example, a DSP block in Xilinx FPGAs consists of a 25-bit x 18-bit multiplier followed by an accumulator; thus potentially every DSP can be instantiated as a MAC unit to perform 16-bit fixed-point multiplications or 32-bit floating-point multiplications [3] and accumulation. In this case, assuming each DSP can be used as a MAC unit, we calculate the potential peak throughput in terms of the number of DSPs instantiated in the design:

$$\text{Potential Peak Throughput} = 2 \times \#\text{DSP} \times f \tag{28}$$

where *#DSP* is the number of DSPs after synthesis and implementation. Thus the DSP utilization efficiency is defined as:

$$\text{DSP Util. Efficiency} = \frac{\text{Eff. Throughput}}{\text{Potential Peak Throughput}} \tag{29}$$

Note that any techniques that synthesize multipliers by LUT or use low precision operands, such as 8-bit or 4-bit numbers, to produce multiple products out of a single DSP should result in higher DSP utilization efficiency.

4.3.5 Power. We measured the wall-plug power using a Voltcraft 4500ADVANCED Energy Monitor (Fig. 15).

As shown in Fig. 15, the power of the baseboard plus the fan when the Zynq MMP is not mounted is 7.9 W. It also includes the power of the power supply. It increases to 9.4 W when the Zynq MMP is mounted without programming the FPGA so that the Zynq MMP is in idle state. After programming the FPGA, the power rises to 10.0 W and further to 15.2 W when the system is iteratively

[3] A single-precision (32-bit) floating-point multiplier can be synthesized using multiple DSPs to achieve high performance or single DSP at the expense of potentially longer latency and higher LUT usage.

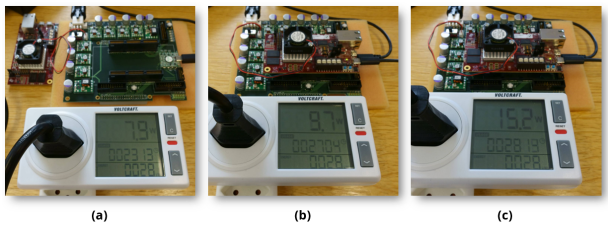

Figure 15: (a): Measured power of the baseboard plus the fan; (b) measured power of the whole system with the Zynq MMP in idle; (c) measured power of the whole system when DRNN is fully loaded by iteratively calculating GRU layer outputs

calculating the hidden layer outputs. Thus the power consumption of the Zynq MMP is 15.2 − 7.9 = 7.3 W and the on-chip power, i.e, the power of the FPGA PL, is 15.2 − 9.7 = 5.5 W. This number is nearly identical to 5.503 estimated from the Xilinx Power Analyzer using a switching rate of 50%. The power analyzer shows that the power of DRNN and weight BRAM are respectively 2.727 W and 2.284 W. The measured power consumption is summarized in Table 5.

Table 5: Power consumption hierarchy of the system

	Power (W)
Total (Running DRNN)	15.2
Baseboard + Fan	7.9
Baseboard + Fan + Zynq MMP	9.7
Zynq MMP	7.3
FPGA	5.5

4.3.6 Power Efficiency. When the delta threshold is 0x80, the effective power efficiency of the DRNN is 1198/5.5 = 218 GOp/s/W with respect to the FPGA on-chip power or 1198/7.3 = 164 GOp/s/W with respect to the power of the Zynq MMP board. A summary of the DRNN performance is shown in Table 6.

Table 6: Summary of DRNN Performance

Parameter	Value
Delta Threshold	0.5 (0x80)
Frequency	125 MHz
Effective Throughput	1198 GOp/s
Potential Peak Throughput	192 GOp/s
MAC Computation Efficiency	623.96%
Power Efficiency (FPGA)	217.82 GOp/s/W
Power Efficiency (Board)	164.11 GOp/s/W

4.4 Comparison to Prior Work, CPU & GPU

Table 7 shows performance of DRNN compared to prior work, an Intel i7-8700K CPU, and an NVIDIA GTX 1080 Ti GPU. The total power of the Zynq MMP development board measured by the wall plug power meter is used in this comparison. Ref. [10] supports either 32-bit floating-point or 16-bit fixed-point precision and we use results obtained in fixed-point precision. Ref. [4] has three different implementations, and we used the one called "DeepStore" that stores weights on BRAM in Table 7, which is the closest one to our work. To measure the performance of CPU/GPU on the same task discussed in Section 4.3, we try to follow a method similar to the one used in Ref. [13]. For the CPU benchmark, we run the test model in Theano with Intel Software Optimization (Intel MKL 2018), and the CPU power is measured by using the Stress Terminal UI monitoring tool[4] since our CPU does not support the Intel pcm-power utility. For GPU, the model is run in Theano with CUDA 9 and cuDNN 7 while the GPU power is measured using the nvidia-smi utility. With batch size = 128, the CPU/GPU finished the task in 295.87/45.05 ms with 35.6/95.9 W average power. On this task, DRNN is 63.8x/9.7x faster and 312.7x/128.5x more power efficient than CPU/GPU.

5 CONCLUSION

In this paper, we illustrate how high power efficiency can be achieved for RNN inference by combining the DN algorithm, of which the training process is already integrated in Lasagne powered by Theano, with our proposed DRNN hardware architecture. The DRNN features the ability to reduce MxV operations and corresponding weight fetches by skipping dispensable neuron activation changes below a threshold. The DRNN allows trade-off between accuracy and runtime cost. Our previous research on the DN algorithm [25] showed that pre-trained RNNs can also be accelerated by simply running them as delta networks, but the accuracy will be higher if the RNN is trained as a delta network.

The DRNN is implemented on an Xilinx Zynq-7100 FPGA embedded on a Zynq MMP development board running at 125 MHz. Without weight compression and using 16-bit parameters, the DRNN achieved an effective throughput of 1.2 TOp/s and MAC computation efficiency of 623.96% with negligible numerical accuracy loss in a simple speech RNN. The power consumption of the Zynq MMP board with a programmed DRNN is 7.3 W, leading to a power efficiency of 164 GOp/s/W. The use of pruning and other optimizations allowed Han et. al [13] to achieve 2520 GOp/s, which is impressive considering they used external DRAM for weights. Chang et. al [4] proposed an implementation using BRAM for weights, but achieved 0.45 GOp/s/W. The power efficiency achieved by DRNN shows the utility of the delta network approach even without any other optimizations.

The main limitation of our work is the low scalability limited by the available number of BRAM blocks. Moreover, to provide sufficient bandwidth, space in BRAM is not fully utilized. In the future, we will address these limitations to target a lower power design with better scalability to process large multi-layered RNNs.

ACKNOWLEDGMENTS

This work was supported by Samsung Advanced Institute of Technology (SAIT), University of Zurich and ETH Zurich. We thank A. Rios-Navarro, R. Morales and A. Linares-Barranco from the University of Seville for creating the baseboard. We also thank A. Aimar,

[4] https://github.com/amanusk/s-tui

Table 7: Comparison to prior works and CPU, GPU

	This work	[13]	[10]	[4]	CPU	GPU
Hardware Model	**XC7Z100**	XCKU060	GSMD5	XC7Z045	i7-8700K	GTX 1080 Ti
Frequency [MHz]	**125**	200	150	142	-	-
Input Precision	**Fixed16**	Fixed16	Fixed16	Fixed16	Float32	Float32
Weight Precision	**Fixed16**	Fixed12	Fixed16	Fixed16	Float32	Float32
#DSP	**768**	1504	1036	-	-	-
#Multiplier	**768**	1536	-	-	-	-
#MAC	**768**	1024	-	-	-	-
Peak Throughput [G(FL)Op/s]	**192**	409.6	-	-	-	-
Effective Throughput [G(FL)Op/s]	**1198**	2520	315.85	1.04	18.78	123.34
MAC Comp. Efficiency	**623.96%**	615.23%	-	-	-	-
DSP Util. Efficiency	**623.96%**	418.88%	101.62%	-	-	-
Power [W]	**7.3**	41	25	2.3	35.6	95.9
Power Efficiency [G(FL)Ops/s/W]	**164.11**	61.46	12.63	0.45	0.53	1.29

E. Calabrese and other Sensors Group members for support on the FPGA implementation.

REFERENCES

[1] A. Aimar, H. Mostafa, E. Calabrese, A. R. Navarro, R. T. Morales, I-A. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S-C. Liu, and T. Delbrück. 2017. NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps. *CoRR* abs/1706.01406 (2017). arXiv:1706.01406

[2] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. Engel, L. Fan, C. Fougner, T. Han, A. Y. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Y. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu. 2015. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. *CoRR* abs/1512.02595 (2015).

[3] N. Bell and M. Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA.* NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.

[4] A. X. M. Chang and E. Culurciello. 2017. Hardware accelerators for Recurrent Neural Networks on FPGA. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS) (ISCAS '17).*

[5] A. X. M. Chang, B. Martini, and E. Culurciello. 2015. Recurrent Neural Networks Hardware Implementation on FPGA. *CoRR* abs/1511.05552 (2015).

[6] K. Cho, B. v. Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014).

[7] R. Dorrance, F. Ren, and D. Marković. 2014. A Scalable Sparse Matrix-vector Multiplication Kernel for Energy-efficient Sparse-blas on FPGAs. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA '14).* ACM, New York, NY, USA, 161–170.

[8] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt. 2014. A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines (FCCM '14).* IEEE Computer Society, Washington, DC, USA, 36–43.

[9] A. Graves, A. Mohamed, and G. E. Hinton. 2013. Speech Recognition with Deep Recurrent Neural Networks. *CoRR* abs/1303.5778 (2013).

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

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

[12] K. Guo, S. Han, S. Yao, Y. Wang, Y. Xie, and H. Yang. 2017. Software-Hardware Codesign for Efficient Neural Network Acceleration. *IEEE Micro* 37, 2 (Mar 2017), 18–25.

[13] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17).* ACM, New York, NY, USA, 75–84.

[14] S. Han, H. Mao, and W. J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015).

[15] S. Han, J. Pool, J. Tran, and W. J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS'15).* MIT Press, Cambridge, MA, USA, 1135–1143.

[16] S. Hochreiter and J. Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780.

[17] M. Horowitz. 2014. 1.1 Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC).* 10–14.

[18] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. 2016. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *CoRR* abs/1609.07061 (2016). arXiv:1609.07061

[19] D. Kadetotad, S. Arunachalam, C. Chakrabarti, and Jae sun Seo. 2016. Efficient memory compression in deep neural networks using coarse-grain sparsification for speech applications. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* 1–8.

[20] S. Kapur, A. K. Mishra, and D. Marr. 2017. Low Precision RNNs: Quantizing RNNs Without Losing Accuracy. *CoRR* abs/1710.07706 (2017). arXiv:1710.07706

[21] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung. 2016. FPGA-Based Low-Power Speech Recognition with Recurrent Neural Networks. *CoRR* abs/1610.00552 (2016).

[22] I. McGraw, R. Prabhavalkar, R. Alvarez, M. G. Arenas, K. Rao, D. Rybach, O. Alsharif, H. Sak, A. Gruenstein, F. Beaufays, and C. Parada. 2016. Personalized speech recognition on mobile devices. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* 5955–5959.

[23] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. 2010. Recurrent neural network based language model.. In *Interspeech*, Vol. 2. 3.

[24] A. H. Namin, K. Leboeuf, R. Muscedere, H. Wu, and M. Ahmadi. 2009. Efficient hardware implementation of the hyperbolic tangent sigmoid function. In *2009 IEEE International Symposium on Circuits and Systems.* 2117–2120.

[25] D. Neil, J. H. Lee, T. Delbruck, and S-C. Liu. 2017. Delta Networks for Optimized Recurrent Network Computation. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 2584–2593.

[26] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr. 2016. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL).* 1–4.

[27] J. Schmidhuber. 2014. Deep Learning in Neural Networks: An Overview. *CoRR* abs/1404.7828 (2014).

[28] Xilinx. 2017. *Zynq-7000 All Programmable SoC Data Sheet: Overview.* https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[29] L. Zhuo and V. K. Prasanna. 2005. Sparse Matrix-Vector Multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays (FPGA '05).* ACM, New York, NY, USA, 63–74.