OCEAN: An On-Chip Incremental-Learning Enhanced Artificial Neural Network Processor with Multiple Gated-Recurrent-Unit Accelerators

Chixiao Chen, *Member, IEEE*, Hongwei Ding, Huwan Peng, Haozhe Zhu,
Yu Wang, *Member, IEEE* and C.-J. Richard Shi, *Fellow, IEEE*

***Abstract*—The paper presents OCEAN: an artificial neural network processor designed for accelerating gated-recurrent-unit (GRU) inference and on-chip incremental learning for sequential modeling. Three energy-quality-scalable training methods, including analytic, numerical and semi-analytic schemes are investigated. Implemented in 65nm CMOS with silicon area of 2.9×3.5 mm^2, the OCEAN processor features a 32-bit RISC core, 64KB on-chip SRAM, and eight 16-bit GRU accelerators for inference and gradient computation. Each accelerator is optimized for RNN data flow with five pipeline stages, and enhanced for efficient gradient computation. The processor is measured to consume 155mW at the peak clock rate of 400MHz and the supply of 1.2V or 6.6mW at 20MHz/0.8V for speech and biomedical applications. Both inference and on-chip incremental learning are accomplished on well-known AI tasks such as handwritten digits recognition, semantic natural language processing, and biomedical waveform based seizure detection.**

***Index Terms*—Recurrent neural network (RNN), gated recurrent unit (GRU), inference, on-chip training, deep learning processor, energy-efficient accelerator, gradient computing.**

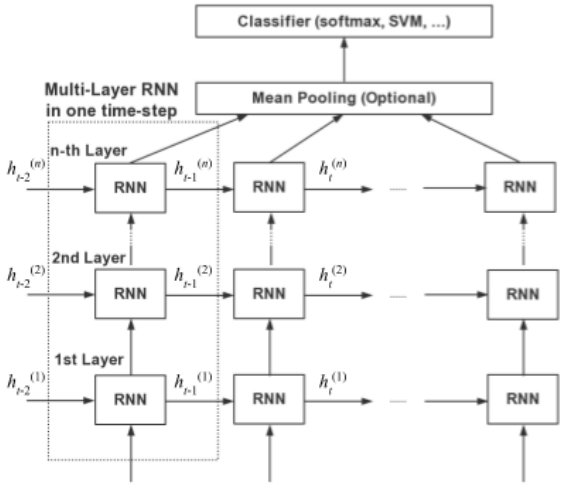


Fig. 1. A classifier using recurrent neural networks unfolded in time.

I. Introduction

RECENT success of deep learning in artificial intelligence (AI) has intensified design of application-specific integrated circuits (ASICs) for accelerating neural network computation. Out of various deep learning applications [1], there are two major categories of neural networks, convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Much work has been directed towards accelerating CNNs [2]- [6], but little to RNNs. While CNNs are known to be effective for image and video applications, RNNs, especially long short term memory (LSTM) [19] and gated recurrent unit (GRU) [20], remain the dominating networks for such applications including speech recognition and nature language processing (NLP) [10], and biomedical signal processing, where sequential modeling is essential. In this paper, we present **OCEAN** — an On-Chip incremental-learning Enhanced Artificial Neural network processor, th first processor designed specifically for accelerating RNNs, more specifically GRUs, for power-constrained sequential modeling on edge devices [9].

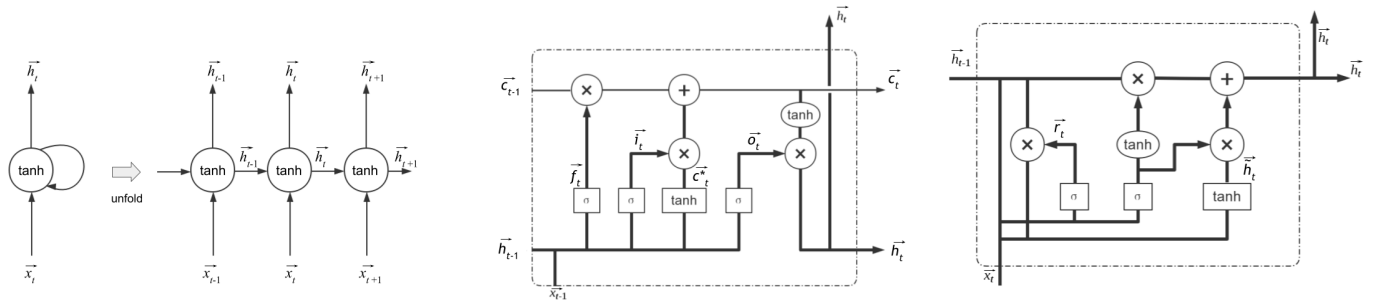
OCEAN has two major contributions to the state of art. Firstly, OCEAN employes a set of techniques to reduce the inherient hardware complexity associated with RNN implementation. We observe that most time a local neuron (cell) is interacting with its four neighbouring neurons (left, right, upper, lower), therefore a fully-connected 4-cell GRU is designed and optimized as a core. An additional consideration is that digital logic implementation can be adequately optimized while the gate fanout and fanin numbers are limited to 4 [14]. The OCEAN processor contains eight of such 4-cell GRU cores. A RISC controller programable in the C lanuguage is used to control how to use these eight 4-cell GRU cores to construct large-size fully or sparsely connected GRUs in a time-multiplexed or space-multiplexed manner. To allow a 4-cell GRU core to be used efficiently to construct large GRUs with arbitrary number of cells, the registers for bias vectors are enhanced for accumulating partial sums. In addition, a limited input fixed point (LIFP) architecture is designed to support high-accuracy fixed point hardware implementation of multiplication-and-accumulation (MAC) operation, the most important operation in deep learning.

Secondly, the OCEAN processor supports effective on-chip incremental learning; i.e., localized training on edge devices with affordable power and silicon area. Normally, training of

Manuscript received January 9, 2018. A priminary version of this paper was presented at the *43rd European Solid-State Circuits Conference (ESSCIRC), Leuven, Belgium, Sept. 11-14, 2017* [9].

C. Chen, H. Ding, H. Peng and C.-J. Richard Shi are with the Department of Electrical Engineering, University of Washington, Seattle WA 98195, USA. This work was done while the authors were with the Institute of Brain-Inspired Circuits and Systems (IBICAS), Fudan University, Shanghai, China.

H. Zhu and Yu Wang are with the Institute of Brain-Inspired Circuits and Systems (IBICAS), Fudan University, Shanghai, China.



$$\vec{h}_t = \tanh(\mathbf{W}_x\vec{x}_t + \mathbf{W}_h\vec{h}_{t-1} + \vec{b})$$

(a)

$$\vec{f}_t = \sigma(\mathbf{W}_{xf}\vec{x}_t + \mathbf{W}_{hf}\vec{h}_{t-1} + \vec{b}_f)$$
$$\vec{i}_t = \sigma(\mathbf{W}_{xi}\vec{x}_t + \mathbf{W}_{hi}\vec{h}_{t-1} + \vec{b}_i)$$
$$\vec{o}_t = \sigma(\mathbf{W}_{xo}\vec{x}_t + \mathbf{W}_{ho}\vec{h}_{t-1} + \vec{b}_o)$$
$$c_t^\star = \tanh(\mathbf{W}_{xc}\vec{x}_t + \mathbf{W}_{hc}\vec{h}_{t-1} + \vec{b}_c)$$
$$\vec{c}_t = \vec{f}_t \odot \vec{c}_{t-1} + \vec{i}_t \odot c_t^\star$$
$$\vec{h}_t = \vec{o}_t \odot \tanh(\vec{c}_t)$$

(b)

$$\vec{r}_t = \sigma(\mathbf{W}_{xr}\vec{x}_t + \mathbf{W}_{hr}\vec{h}_{t-1} + \vec{b}_r)$$
$$\vec{z}_t = \sigma(\mathbf{W}_{xz}\vec{x}_t + \mathbf{W}_{hz}\vec{h}_{t-1} + \vec{b}_z)$$
$$\tilde{\vec{h}}_t = \tanh(\mathbf{W}_{xh}\vec{x}_t + \mathbf{W}_{hh}(\vec{r}_t \odot \vec{h}_{t-1}) + \vec{b}_h)$$
$$\vec{h}_t = \vec{z}_t \odot \vec{h}_{t-1} + (1 - \vec{z}_t) \odot \tilde{\vec{h}}_t$$

(c)

Fig. 2. Common recurrent neural network topologies (a) vanilla model (b) LSTM model (c) GRU model.

neural networks is implemented by power-hungry GPUs due to the critical precision requirement and non-uniform operations [11]. Existing work on deep-learning ASICs is mainly focused on inference. However, with the increasing demand on data security and bandwidth efficiency, edge devices with on-chip training are desirable. A mobile speech assistant, for example, requires to adapt its accuracy with a user's voice by on-chip learning. A deep learning processor capable of on-chip learning was proposed for CNNs [12]. It uses four deep learning cores dedicated for training. This requries large silicon area. In this paper, we introduce an on-chip learning architecture based on semi-analytic gradient computation. We show both empirically and analytically that this architecture can reuse limited-precision interference hardware to achieve sufficient accuracy adequately for on-chip incremental learning.

The rest of the paper is organized as follows. Section II first reviews the RNN basics, and then introduces RNN's hardware sharing strategy. On-chip incremental learning with various methods of gradient computing is studied in Section III. Section IV presents the architecture and implementation of the OCEAN processor. Section V shows several programming examples. Measurement results are described in Section VI. Section VII concludes the paper.

II. RNNs and Hardware Implementation

Different from CNNs, an RNN features its temporal behavior by a recurrent path from the previous neurons' outputs to inputs of the next time step. Figure 1 illustrates a typical RNN classifier unfolded in the temporal domain. Each RNN unit's result, $\vec{h}_t^{(i)}$, is generated by the input of t-th time step, $\vec{x}_t$ and the RNN results of $(t-1)$-th time step, namely, $\vec{h}_{t-1}^{(i)}$, where $i = 1, 2, \cdots, n$ stands for the i-th layer. In this way, the RNN's weights are reused for the entire sequential inputs. After the recurrent layers, the output layer includes an optional pooling layer and a fully connected network (FCN).

TABLE I
Parameter Comparison between RNNs

	Vanilla	LSTM	GRU
Gate Coefficient	0	3	2
Recurrent Out	1	2	1
Mtrx.-Vt Prdt	2	8	6
Bias Paramt.	1	4	3
Weight Paramt.	$MN+N^2$	$4(MN+N^2)$	$3(MN+N^2)$
Activation	N	$5N$	$3N$
Bit-wise Prdt	0	$3N$	$3N$
MAC total	$MN+N^2$ $+N$	$4(MN+N^2)$ $+8N$	$3(MN+N^2)$ $+6N$

$\vec{x}_t = [x_1, x_2, \cdots, x_M]^T \in \Re^M$, $\vec{h}_t = [h_1, h_2, \cdots, h_N]^T \in \Re^N$

Figure 2 illustrates three most common RNN unit models: (a) vanilla RNNs, (b) long short term memory (LSTM) [19], and (c) gated recurrent units (GRUs) [20]. Unlike vanilla RNNs, LSTMs and GRUs do not suffer from the gradient explosion and vanishing issues [18], and have been widely used for deep learning. In Fig. 2, element-wise product is denoted by $\odot$. Gate coefficients, $\vec{f}_t, \vec{i}_t, \vec{o}_t$ in LSTMs and $\vec{r}_t$, $\vec{z}_t$ in GRUs, are element-wisely multiplied by recurrent outputs $\vec{c}_t, \vec{h}_t$ in LSTMs and $\vec{h}_t$ in GRUs. Recurrent outputs from last time step are c_{t-1} and h_{t-1}. Various $\mathbf{W}$ matrices represent weights and vectors $\vec{b}$ are *biases*. Two non-linear activation functions sigmoid and tanh are defined as

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}, \quad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \qquad (1)$$

Fig. 3(a) shows the data flow diagram of GRU computation as in Fig. 2(c). We assume that the MAC operation is taking about the same time as the activation function. Then the critical path for GRU computation is from $\vec{r}_t$, $\tilde{h}_t$ to $\vec{h}_t$, indicated by the dotted arrow in Fig. 3(a). This can be partioned to five stages with approximately the same delay as shown in Fig. 3(b), where the main computing blocks include MAC

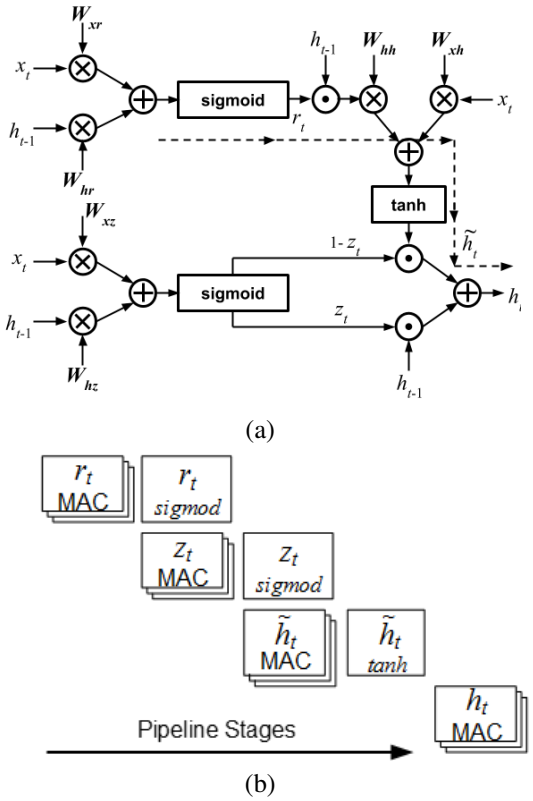


Fig. 3. (a) GRU' data flow, and (b) hardware sharing pipeline strategy.

blocks and activation blocks, which can be pipelined. Note that both MAC and activation blocks are reused by $\vec{r_t}$ and $\vec{z_t}$. The hardware sharing strategy has no throughput penalty, because the critical path, from $\vec{r_t}$, $\vec{\tilde{h}_t}$ to $\vec{h_t}$, maintains.

Table I summarizes the complexities of the three RNN networks, where input $\vec{x}_t$ is an M-dimension column vector and output $\vec{h}_t$ is an N-dimension column vector. All the weight matrices in Fig. 2 have sizes of $M \times N$. The total multiplication-and-accumulation (MAC) operation number is obtained by summing the numbers of weight-input product, activation, and element-wise product, where one activation is counted as one MAC. The last row shows the MAC requirement adopting five-stage pipelined architecture.

III. On-Chip Incremental Learning

In this section, we first review the gradient descent based RNN training process, and then describe three methods of gradient computation: analytic, numerical, and semi-analytic. These three methods are then compared in terms of their accuracy, bit precision, complexity. Finally on-chip implementation of semi-analytic gradient computation is described.

A. Gradient Descent based Training

We are interested in classifying a *sample* consisting of a *seqeunce* of inputs $\vec{x_1}, \vec{x_2}, \cdots, \vec{x_T}$ into one of K classes. Let $\hat{y_k}$ is the one-hot ground truth label (0 or 1) and k is the class index. The training process seeks to minimize a loss function E which is often defined by a cross-entropy function

$$E = \sum_{k=1}^{K} -\hat{y_k} \cdot \log(y_k), \tag{2}$$

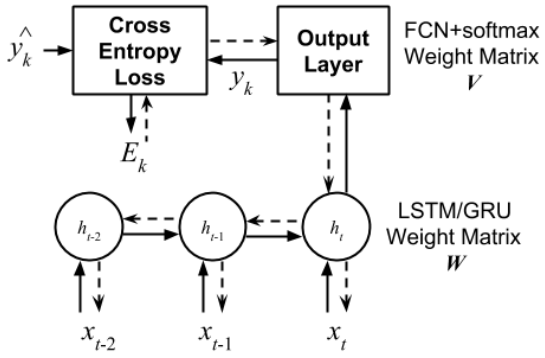


Fig. 4. Back propagation through time for RNN training.

where y_k is the probabily of being classified in class k defined by a softmax function

$$y_k = \text{softmax}(v_k) = \frac{\exp(v_k)}{\sum_{i=1}^{N} \exp(v_i)}, \ k = 1, 2, \cdots, K \tag{3}$$

As illustrated in Fig. 1, the predicted scores $\vec{v_k}$ are obtained by cascading RNNs and FCNs, whose weights are denoted as matrix $\mathbf{W}$ and $\mathbf{V}$ respectively, shown in (4).

$$\vec{v_k} = \mathbf{V} \cdot \vec{h_T} = \mathbf{V} \cdot \text{RNN}(\vec{x_1}, \vec{x_2}, \cdots, \vec{x_T} | \mathbf{W}), \tag{4}$$

where $\text{RNN}()$ represents the LSTM/GRU function, and $\vec{h_T}$ is the recurrent output of the final time step and T is the index of last time step.

The problem of training a neural network is to find the weights to minimize the loss function E. This is done by gradient descent (GD). GD is a procedure of repeatedly evaluating the loss function gradient and then performing a weight update along the direction of gradients towards the minimum loss. For large data sets of millions of samples, it is usual to compute the loss function gradient over multiple samples, referring to as a stochastic gradient descent (SGD) or *mini-batch* gradient descent. The entire training set is divided into multiple subsets by a batch size, m and processed as,

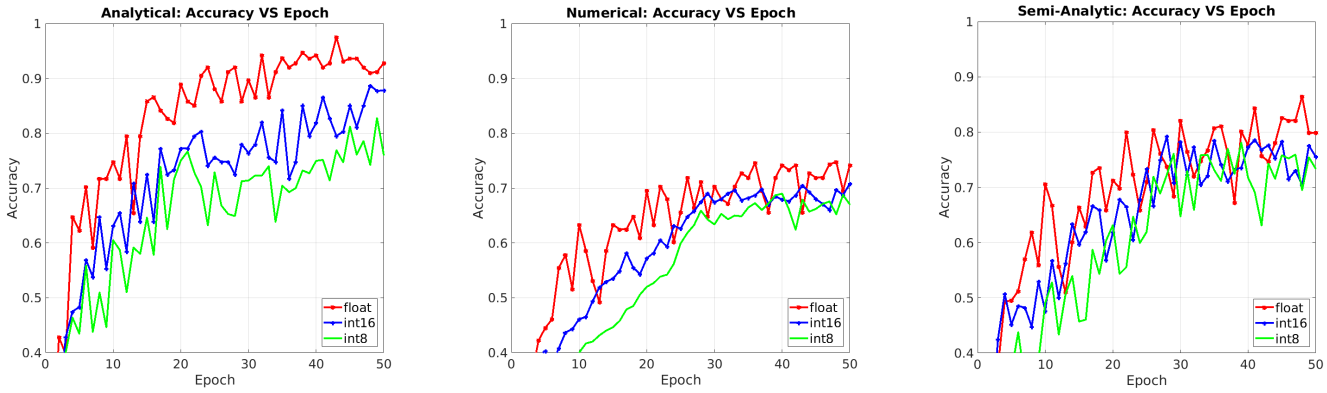
$$\vec{w}^{\,(j+1)} = \vec{w}^{\,j} - \eta \cdot \frac{1}{m} \sum_{i=1}^{m} \frac{\partial E_i}{\partial \vec{w}^{\,j}} \tag{5}$$

where $\vec{w}^{\,j}$ is a weight vector of j-th batch of the training process, η is the step size defined by various algorithms [17], and E_i is the error of the i-th sample in the m-sized batch.

Moreover, updating the weights within a single pass of the training set is likely not sufficient. An iterative process, which passes the training set multiple times, is often adopted. Each time the set is gone through is called an *epoch*. The training process normally needs tens of epochs to reach the final convergence.

B. Gradient Computation

RNN gradient computing requires propagation through time, as shown in Fig. 4. In each sample, the RNN goes through T time steps, referring as to the *sequence length*. To obtain the loss function gradient, the derivative chain rule along with the dashed line in Fig. 4 are accumulated in the temporal domain. Now we describe three methods for gradient computation: analytic method, numerical method, and semi-analytic method.



(a) (b) (c)

Fig. 5. RNN training accuracy by different gradient computing methods: (a) analytic (b) numerical (c) semi-analytic.

1) Analytic Method: Substituting (4) into (2) and taking the derivative, the loss gradient can be obtained analytically

$$\frac{\partial E}{\partial \vec{w}} = \underbrace{\frac{\partial E}{\partial \vec{v_k}}}_{\text{Out Layer}} \cdot \underbrace{\frac{\partial \vec{v_k}}{\partial \vec{w}}}_{\text{DNN}} = \underbrace{(\vec{y_k} - \hat{y_k})^T}_{\text{softmax gradient}} \cdot \frac{\partial \vec{v_k}}{\partial \vec{h_T}} \cdot \frac{\partial \vec{h_T}}{\partial \vec{w}}, \quad (6)$$

where superscript T represents the transpose of a vector, and the subscript T denotes the last time step.

For RNNs, the standard back-propagation method evolves to back propagation through time (BPTT), because all the weight performs on each time step accumulatively [19]. The gradients of an RNN, $\partial \vec{h_t}/\partial \vec{w}$, can be expressed as

$$\frac{\partial \vec{h_t}}{\partial \vec{w}} = \sum_{p=0}^{t} \left(\prod_{q=p+1}^{t} \frac{\partial \vec{h_q}}{\partial \vec{h_{q-1}}} \right) \frac{\partial \vec{h_p}}{\partial \vec{w}}, \quad (7)$$

where $\vec{h_{q-1}}$ is an input of $\vec{h_q}$ per the RNN definition.

2) Numerical Method: A straighforward numerical method to approximate gradients is to compute the differnce of the loss function value numerically

$$\frac{\partial E}{\partial \vec{w}} = \sum_{i=1}^{m} \hat{y_{k'}} \frac{\log(y_{k'}|_{w_0}) - \log(y_{k'}|_{w_0+\Delta w})}{m \cdot \Delta w}, \text{ when } \vec{w} = w_0 \quad (8)$$

where Δw is a tiny incremental step, m is the batch size and $y_{k'}$ is the classifier result, i.e. the outputs of the *softmax* regression of the k-th class. $\hat{y_{k'}}$ is the one-hot label, and k' is the labeled class index. The motivation for using numerical gradient is that $y_{k'}$ in (8) can be calculated by reusing the RNN inference core.

3) Semi-Analytic Method: This method adopts the analytical gradient for softmax and the numerical gradient for the neural networks. In this case, (6) is transformed to

$$\frac{\partial E}{\partial \vec{w}} = \sum_{k=1}^{K} (\vec{y_k} - \hat{y_k}) \cdot \left(\sum_{i=1}^{m} \frac{\vec{v_k}|_{w_0+\Delta w} - \vec{v_k}|_{w_0}}{m \cdot \Delta w} \right) \quad (9)$$

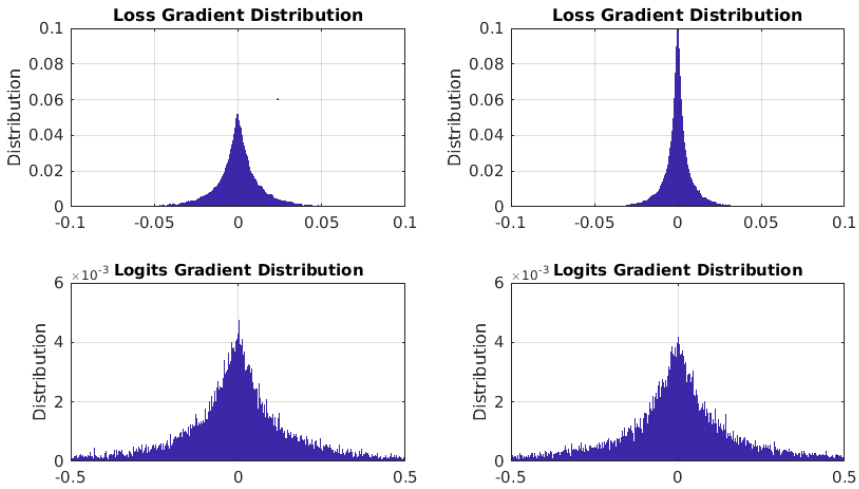
where $\vec{v_k}$ is the k-th score of the neural network and K is the total output class number, Δw is the incremental step, and m is the batch size. In contrast with the numerical method, the semi-analytic method reuses the inference core to compute the DNN gradients only, whereas the softmax gradients are obtained as the analytic method.

C. Accuracy and Bit-Precision of Gradient Computation

To evaluate the EQ scalable performance of different gradient computation methods, we perform them on the RNN/FCN training of the MNIST, a well-known handwritten digit database [26]. The RNN is a 32-input 128-cell GRU network, namely $M = 28, N = 128$ in Table I. The softmax classifier involves a 128×10-matrix FCN. The training process adopts the mini-batch stochastic gradient descent scheme, where the batch size is 128. The training process here is implemented in Tensorflow. The analytic gradient computation are realized by the standard optimizer. The rest two gradient computing methods are performed as (8) and (9) indicate respectively. Note that TensorFlow also has three built-in number representations to use: 16-bit floating-point (FP16), 16-bit integer (INT16) and 8-bit integer (INT8). We also implement numerical gradient computation, based on and semi-analytic gradient computation , based on using FP16, INT16 and INT8 number representations. There is no explicit normalization layer in the Tensorflow code.

Figure 5 shows how the training accuracy (Accuracy) improves over the increase of the training set (Epoch size), for three gradient computing methods with respect to three number representations. The accuracy is *hit* rate in the validation set when the class with maximum computed probability is consistent with the labeled class.

From Fig. 5, we have the following observations, (i) The analytic method demonstrates that the final validation accuracy is sensitive to the number representation. The peak validation accuracy for 16-bit floating-point (FP16), 16-bit integer (INT16) and 8-bit integer (INT8) are 97.5%, 88.7%, and 82.8% respectively. The INT16 representation loses almost 10% accuracy compared with FP16, (ii) The overall accuracy of the numerical method is worse than the analytic method, but the differences among different number format in the numerical method are less. Compared with the analytic method, the peak accuracy of the numerical method is 75%, 70.5% and 68.9% for FP16, INT16, and INT8 respectively, (iii) The semi-analytic method trades-off the two previous methods in both accuracy and number format sensitivity. The peak accuracy of the numerical method is 84.3%, 79.2% and 78.1% for FP16, INT16, and INT8 respectively. An intuition on the improve-



(a) (b)

Data Set		Before Convg.	After Convg.
IMdB	Loss grad std.	0.0354	0.0066
	Logitis grad std.	0.1599	0.1690
	Accur.	55%	81%
MNIST	Loss grad std .	0.170	0.0100
	Logits grad std.	0.2501	0.2784
	Accur.	27%	93%
F-MNIST	Loss grad std.	0.1042	0.0120
	Logits grad std.	0.2630	0.2722
	Accur.	19%	89%

(c)

Fig. 6. RNN gradient distribution at (a) IMdB data set beginning phase (b) completion phase (c) the standard deviation trend among different data set.

ment of the semi-analytic method is the softmax gradient term in (6) is the one shrinks the most during the convergence. Computing it analytically relaxes its representation range of various number formats.

It can be interpreted as the precision difference for various gradient computing schemes. Figure 6 demonstrates the actual gradient distribution at the different phases of training, (a) at the beginning phase and (b) at the phase after complete convergence. According to (6), we compared the overall loss gradient ($\partial E/\partial w$) and the score gradient ($\partial \vec{v_k}/\partial h_t \cdot \partial h_t/\partial w$), also known as *logits* gradient in other literature, before and after the complete training process. Different gradient trends can be observed based on the distribution. First, the overall loss gradient concentrates into zero as the network converges. In the MINIST example, the final distribution has a standard deviation (σ) of 0.01. On the other hand, the score gradient, including both recurrent and fully connected neural network maintains a roughly Gaussian distribution with an average σ of 0.278 throughout the learning process. It is almost 28 times greater than the overall gradient. The simulation is also run across other data sets (IMDB [27] and F-MNIST [28]). Both trends maintain among all the cases, as the table in Fig. 6 (c) illustrates. The starting point of all cases is a 3-epoch pre-trained model rather than the fully random initialization. Empirically, this strategy avoids the initial unnecessary and slow searching phase for the numerical and semi-analytic methods. The score gradient's distribution not only does not shrink after the convergence, but extends slightly according to the standard deviation data among all the three data sets.

The experimental results explain (a) why the training accuracy is more sensitive to number format in the analytic method than the other methods, (b) why the semi-analytic method outperforms the numerical method. It is assumed that the accuracy is determined by how much precisely the gradients can be represented. In the analytic method, INT8 neglects many more near-zero gradients than FP16. Therefore, the accuracy is sensitive to the different number formats. In contrast, the other two methods obtained less precise gradients by approximation. As a result, the final accuracy is limited by the gradient approximation rather than number format. Between the numerical and semi-analytic methods, the score gradient with wider range can be represented more precisely than the overall gradient. In particular, for gradients locating within a range of ± 0.015625 (2^{-7}), a small Δw is not likely to make any difference between two inferences. Therefore, an effective gradient vanishing issue comes up for the numerical gradient computing in the numerical method. This condition is relaxed in the semi-analytic method.

TABLE II
GRADIENT COMPUTATION OPERATION NUMBER COMPENSATION

Method	Multiplication #	Addition #
Analytic	$36(MN+N^2)N^2T$	$24(MN+N^2)N^2T$
Numerical	$(3MN+3N^2+N)^2T$	$9(MN+N^2)^2T$
Semi-Analytic	$N(3MN+3N^2+N)^2T$	$9N(MN+N^2)^2T$

D. Complexty of Gradient Computation

The three methods follow the energy-quality scalable trend. Table II summarizes the number of the computation operations of all the three methods. The conventional analytic method consumes the most computation hardware. Due to the BPTT nature of RNNs, the analytic method requires an entirely different hardware rather than inference core. Though the operation number is acceptable, huge computing power and high bandwidth memory are still mandatory. To the authors' knowledge, the analytic method is only supported by GPU and TPU only now and more desired in the cloud and server. However, edge devices cannot afford its power consumption. Though performing one-time inference through all the weight, the numerical method needs the operation numbers of the same order as the analytic method. More importantly, all the computation hardware in the numerical method is realized by the inference core. However, it achieves the worst accuracy among three.

The hybrid semi-analytic method is a trade-off, or a EQ scalable way between the analytic and numerical methods. It

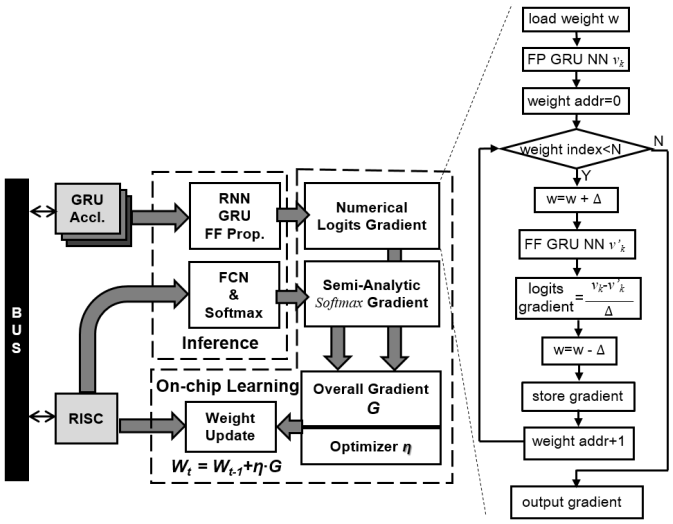


Fig. 7. The inference/on-chip learning flow on the OCEAN processor.

achieves the intermediate accuracy as shown before. Though it consumes most operation (N-times more than numerical methods) as the analytical method, most operations can be realized efficiently by the inference accelerators.

E. On-Chip Implementation of Gradient Computation

Figure 7 proposes an on-chip incrementation of the semi-analytic method based learning. It is assumed that the processor is a system-on-chip platform with a controller core, bus and RNN specific accelerators. In the inference scenarios, the accelerators are responsible for RNN computing, while the core takes the charge of data transfer, FCN and softmax. For incremental learning, a higher level loop repeats performing inference incrementally to obtain all the score gradient, as shown in Fig 7's flow graph. After loading the initial weights, a first RNN feedforward propagation (FP) is performed as a reference score. Then each weight is updated by Δ, and the FP runs again to get the updated score. The score difference is obtained by subtracting the reference score from the updated one. The score gradient is the ratio of the score difference over Δ. Note that normally Δ is a power of 2 to simplify the division into a shift operation. The weights are resumed before the next loop begins. Other operations such as the softmax gradient and weight update are performed by the core.

IV. OCEAN Processor Architecture and Implementation

The overall architecture of the proposed On-Chip incremental learning Enhanced Artificial Neural network processor (**OCEAN**). The OCEAN processor features eight 16-bit gated recurrent unit (GRU) accelerators, a 32-bit Reduced Instruction Set Computing (RISC) core, Direct Memory Access (DMA) module, 64K SRAM, Interconnect-Matrix Memory-Management controller (IMC-MMC), and interface modules interconnected using advanced micro-controller bus architecture (AMBA), where AHB bus connects the RISC core, on-chip SRAM, DMA, and accelerators, and APB bus connects interface modules GPIO, SPI, JTAG and interrupt controller.

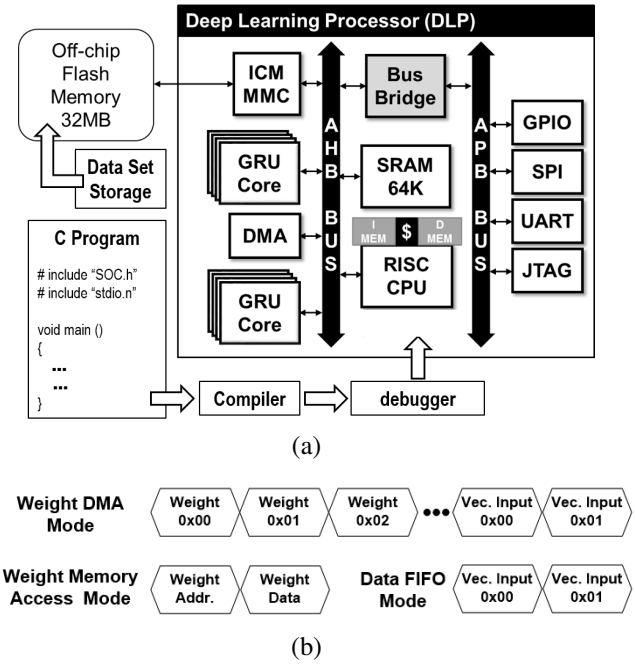


Fig. 8. The proposed **OCEAN** processor (a) overall architecture (b) memory access modes.

A bus bridge connects AHB and APB buses. Eight accelerators are responsible for DNN and gradient computation. The C-programmable RISC core is responsible for how to connect eight accelerators in a time multiplexed and/or space multiplexed manner to construct a large-size fully or sparsely connected RNN, as well as supporting operations other than neural network and gradient computation such as softmax output layer computation.

The memory hierarchy of the OCEAN processor includes the register files of the RISC core and accelerators, a level-one cache of the RISC core, on-chip 64-KB SRAM, and an off-chip volume flash memory. Note that there is no DRAM in the hierarchy, and the 64-KB SRAM is used for main memory access during processing. Initially, all data are stored in the off-chip flash memory and load onto the on-chip SRAM using module IC-MMC.

There are three modes to transfer data between memories, as shown in Fig. 7. The weight direct memory access (DMA) mode copies the full tank of the weight memory to the accelerators from the flash or SRAM modules, and is used at the begining of inference. The bus-interfaced FIFO mode transfers the input/output feature vectors $\vec{x_t}$ and $\vec{h_{t-1}}$. The weight memory access (WMA) mode is designed, in particular, for on-chip incremental learning, It allows individual weightw to be accessible by global address indexing. This allows incremental neural network computation with individually updated weights for numerical or semi-analytic gradient computation.

A. GRU Accelerator for Inference and Gradient Computation

As Section II suggests, the accelerator adopts a hardware sharing pipelined strategy. It includes 5 pipeline stages, illustrated in Fig. 9. The first and third stages are assigned for the matrix-vector product (MVP). Each MVP block has 16 multipliers, and supports a 4×4 matrix multiplying a 4-D

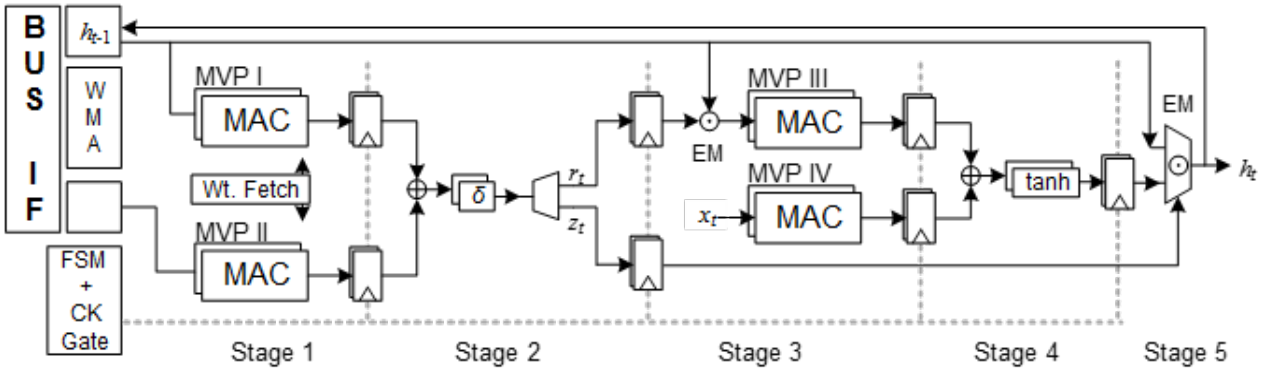


Fig. 9. The proposed 5-stage gated RNN accelerator.

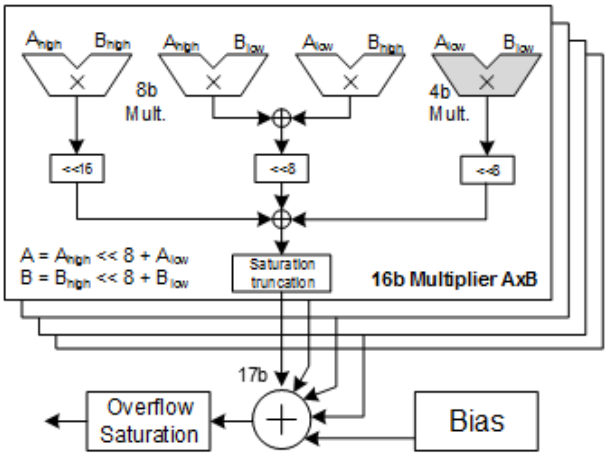
vector simultaneously. The difference between the two MVP stage is that the third stage involves an additional element-wise multiplier (EM) block. The second and fourth stages are assigned for the implementation of the non-linear activation feature. The final stage is another EM and accumulation stage for GRU output. Each accelerator has a 2KB register file to store weights, illustrated as weight memory access (WMA) in Fig. 9.

A finite state machine (FSM) controls the five pipeline stages and generates the gating clocks. For RNNs whose input dimensions are more than 4 ($M > 4$), a partitioning scheme is adopted. All the partial sums are written back to the bias register for next partitioned MVP. An activation operation is performed after all accumulations are completed. The FSM maps the partition scheme onto the MVP stages and control the clock to trigger each pipeline stages. As for low power implementation, the clocks of the MVP output registers in stage 1 and 3 are gated until all MVP operations are done. The final EM stage is triggered after all activations are completed.

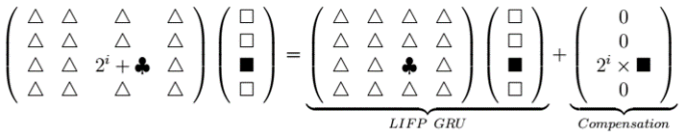
B. Number Format, Multiplier and Overflow Compensation

Recalling from Section III.C, an INT16 presentation demonstrates similar accuracy as the floating points. Therefore, the accelerator adopts a 16-bit fixed point number format, where the MSB is the polarity, the second MSB presents the integer and the rest 14 bit are the fraction bits. To adapt this number format, the pre-trained networks are normalized into a range of $[-2, 2)$.

Based on the above number format, the multipliers in the accelerator adopts three 8-bit fixed-point multipliers, one 4-bit multiplier to achieve a 16-bits fixed-point multiplier, as shown in Fig. 10(a). It is empirically found that the multiplication results need 17 bits to maintain the correctness of RNN algorithms with saturation and truncation. To exploit the hardware saving, we split the full 16 bits into high and low half 8-bit and the products are partitioned into sub-products. The least-significant-bits' computing is approximated by using a compact 4-bit multiplier. The simulation shows the simplification meets the precision requirements for RNNs' inference and gradient computing. Synthesis results show that the approximate design saves 10.4% of area and 16% leakage



(a)



(b)

Fig. 10. (a) MAC design (b) Limited input fixed point overflow compensation.

power than the direct synthesized a 16-bit multiplier at a clock rate of 400MHz without pipelining.

However, the incremental learning process can exceed the range, and thus the overflow fails the learning process. An automatic weight overflow detection and compensation are proposed, known as limited input fixed point (LIFP) compensation and shown in Fig. 10(b). Assuming one large weight coefficient $w_j \in [2, 4)$ and its corresponding input feature x_j, the LIFP scheme detects the possible overflow, sends the $w_j - 2 \in [0, 2)$ part into the MVP block and adds a $x_j << 2$ into the current bias register. Therefore, the accumulative result maintains, i.e., $(w_j - 2)x_j + 2x_j = w_j x_j$. The compensation extends the equivalent bit-width by one bit if only one weight exceeds the range. Note that the process is performed automatically with only one shift operations. There is no extra multiplier needed.

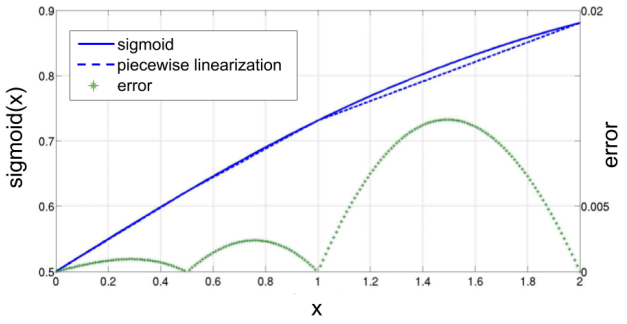


Fig. 11. *Sigmod* approximation and error (amplified by 100x).

C. Nonlinear Activation Functions

Two nonlinear activation functions are used in a GRU accelerator, sigmoid and tanh defined in (1). Both functions contain complicated operations such as division and exponential arithmetic. Several methods have been introduced to approximate these activation functions, including using Taylor expansion approximation [22] and a mini neural network [23]. Here we leverage a three-segment piece-wise linear function to approximate sigmoid. As shown in Fig. 11, this approximation yields the maximum error of 0.0116 and the average error of 0.0044, when the variable x is positive. The negative half is obtained by the symmetry. In the GRU acceleator, the piecewise linear function is realized by a MAC operator, $y = ax+b$, where a and b are parameters using a look-up-table (LUT) according to input value x.

According to (1), the tanh function can be transformed to the sigmoid function as,

$$\tanh(x) = 2\ \mathrm{sigmoid}(2x) - 1 \tag{10}$$

Therefore, tanh is realized by sigmod, shift and subtraction operations.

V. Programming Code Examples

The prototype is programmable by C. The accelerator is configured by the RISC core via registers. A specific mode-configuration data structure, pointed by `*pConfig`, is defined. It includes configuration words of weight write operation mode (DMA mode and memory index access mode), input vector dimensions (M), output vector dimensions (N), sequence length (T), and the number of the utilized cores. The accelerators' results ($\vec{r_t}, \vec{z_t}$ and $\vec{h_t}$) are accessible by loading values of specific registers.

A. RNN Inference Programming

Algorithm 1 lists an example code for RNN inference. The first step of inference is to copy the weights from memory into the accelerator by the DMA mode, as `DMAcopy` in line 2 demonstrates. Before the accelerators calculating, input features are written to the accelerator by the FIFO mode, as `FIFOwrite` in line 8 shows. It is repeated by three loops involved in the inference code to map GRUs of any size. The innermost loop goes through all input feature

Algorithm 1 RNN Inference Algorithm

```
Require: NN & partitioning info, weight, input sequence
Ensure: RNN recurrent output
 1: *pConfig = Full-tank DMA weight write ;
 2: *pData = DMAcopy ( );
 3: // RNN hidden layer
 4: for i = 0; i < N_core /8 ; i=i+8 do
 5:     *pConfig = Reset Data FIFO mode ;
 6:     for j = 0; j < N_seq_length; j++ do
 7:         for k =0; k < N_vec_dim; k++ do
 8:             FIFOwrite(pData[i,j,k]);
 9:         end for
10:     end for
11:     pHidd[i] = pRNN_acc;
12: end for
13: // Fully Connected Layer, Softmax
14: for q = 0; q < N_class ; q++ do
15:     for p = 0; p < N_core ; p++ do
16:         *pClass[q] += *pHidd[p] * FCNwgt[p,q];
17:     end for
18: end for
19: class_result = max_index(pClass , N_class);
20: *pLR = Softmax(pClass , N_class); // optional
```

vectors, where `N_vec_dim` stands for partitioned sub-vector number , namely $M/4$ in Table I, of each input feature. The second loop covers each sample of the entire sequence, where `N_seq_length` represents the sequence length, namely T. By the end of this loop, the output of one GRU cell is completed at the output address of the accelerator, denoted as `pRNN_acc`. The outermost loop defines the times each GRU core is utilized, when overall core usage, `N_core`, is greater than 8. Note that all these `N`-prefixed parameters are also defined in `*pConfig` to configure the FSM. As long as all the weights and features are loaded, the RNN computing kicks off immediately.

The next step is the output layer computation , starting from line 14. The FCN of the output layer is performed by a two-level loop to complete the product between FCN weight matrix `FCNwgt`, namely $\mathbf{V}$ in (4) and RNN output. The top-1 class is returned by a `max_index` function in line 19, which is stored as the classification result, `class_result`. For inference only tasks, the `Softmax` function is not performed because it is not necessary to obtain the exact probability value. However, it is mandatory to run the `Softmax` function in line 20 in the learning tasks, because of the gradient computing in (6).

B. Semi-Analytic Gradient Computing and On-Chip Learning

The on-chip incremental learning program example code is shown in Algorithm 2 box. It is deployed as Fig. 7 implies. Before incremental learning and gradient computing, the inference is performed at first. It is neglected in the example code for simplicity.

As mentioned above, the semi-analytic learning task has two steps. The first is to compute the gradients of neural network weights numerically. The numerical weight gradient computing is computed repeatedly for each weight, as the for loop in line 3 indicates. Setting the weight access mode as RAM, the code deviates one weight by a small fraction, denoted as Δ, as line 4 shows. Afterwards, another inference, defined in the function `RNNinf` is performed in line 6. The gradients are

Algorithm 2 RNN Incremental Learning Algorithm

Require: Initial weight & inference results, input sequence
Ensure: Gradient and new weight

```
1: *pConfig = Weight-RAM mode;
2: // Computer Numerical NN gradient
3: for k = 0; k < N_weight; k++ do
4:     pWgt[k] = pWgt[k] + Δ;
5:     // Do inference with only the labeld class FCN
6:     nClass[class_result] = RNNinf ();
7:     grad_nn[k] = (nClass[class_result] - pClass[class_result]) / Δ;
8:     // Resume weight
9:     pWgt[k] = pWgt[k] - Δ;
10: end for
11: // Compute the labeled class softmax gradient
12: grad_ce = 1 - *pLR[class_result];
13: // SGD Weight Update
14: for k = 1; k < N_weight; k++ do
15:     pWgt[k] += grad_ce * grad_nn[k] >> 4;
16: end for
```

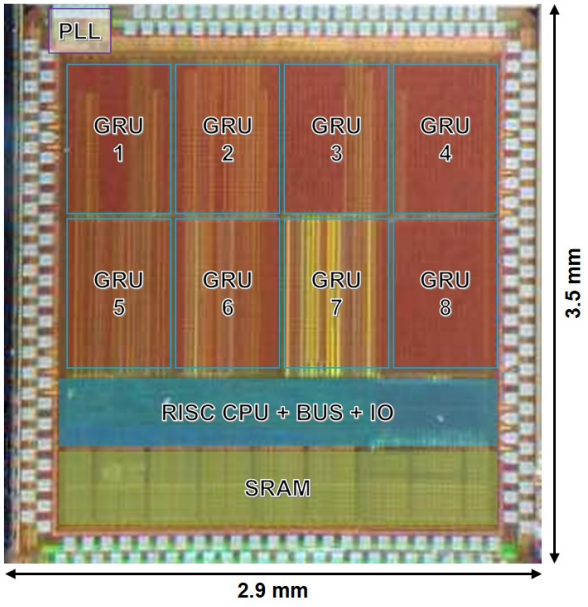


Fig. 12. OCEAN prototype die micrograph and floor plan.

obtained as (9) indicates. Note that Δ is set as the power of 2, i.e., $\Delta = 2^{-Q}$, where Q is a positive integer. Therefore, a shift operation replaces the division. The second step is to compute the analytical $softmax$ gradient by subtracting the generated probability and the ground truth label. For simplicity, we only demonstrate the labeled class subtraction in line 12. Finally, the weights are updated as (5) indicates, demonstrated in line 15. In the code, the learning rate, namely η in (5), is 2^{-4}, realized by another shift operation.

VI. Measurement Results

A silicon prototype of the proposed OCEAN processor has been fabricated in a 65nm CMOS 1P9M technology, integrating 4.91M equivalent logic gates. Its die micrograph is shown in Fig. 12. The RISC core has 150K gates, on-chip 64-KB SRAMs (including the cache) has 1.58M gates and each GRU accelerator has 400K gates. The silicon area is $2.9 \times 3.5mm^2$. An on-chip phase locked loop (PLL) is

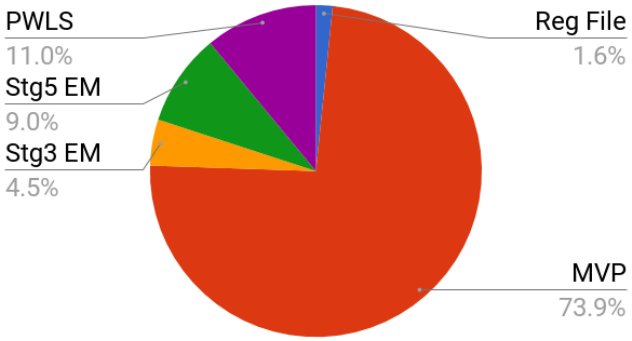


Fig. 13. The area breakdown of a GRU accelerator.

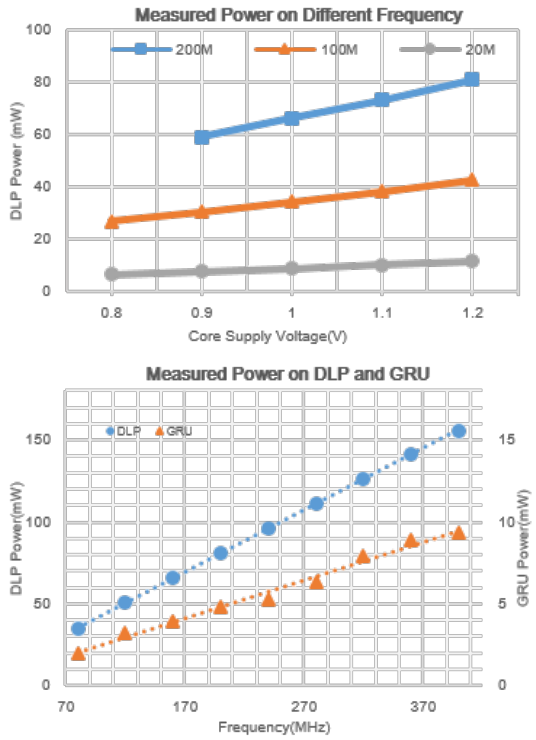


Fig. 14. Power performance of the OCEAN processor.

integrated for local clock generation. The area breakdown of a GRU accelerator is shown in Fig. 13, where arithmetic blocks occupy the most area. The MVP blocks cover almost 3/4 area, two groups of element-wise multipliers count 13.5% in total and all the nonlinear function blocks cover 11.0%. The register file and FSM's area is about 1.6%.

The OCEAN prototype achieves a peak clock rate of 400MHz at 1.2V, consuming 155.8mW in total, and each GRU accelerator consumes 9.36mW. Fig. 14 shows the measured power verse the frequency, and the power verse the supply voltage scaling curves. Overall, the accelerators consume about half of the power, where the rest of the processor including the RISC core consumes the other half. The peak throughput of of all eight accelerators and the RISC core is 311.6 GOPS at 400MHz clock rate, which yields the energy efficiency of 2.0 TOPS/W based on the INT16 number format.

Applications as speech and biomedical engineering are

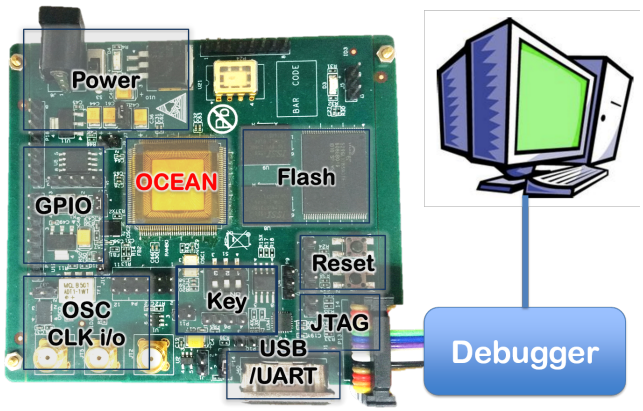


Fig. 15. A validation and demonstration board of the OCEAN processor.

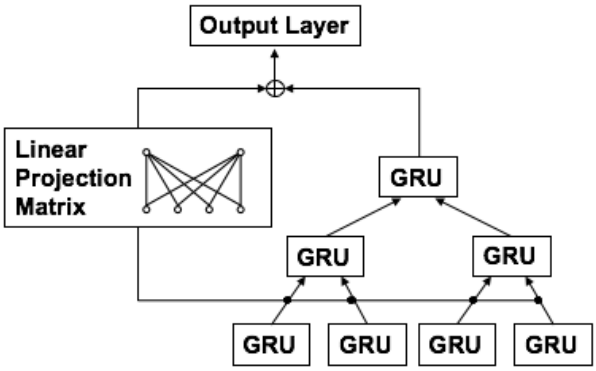


Fig. 16. A 3-layer GRU log-tree residual network.

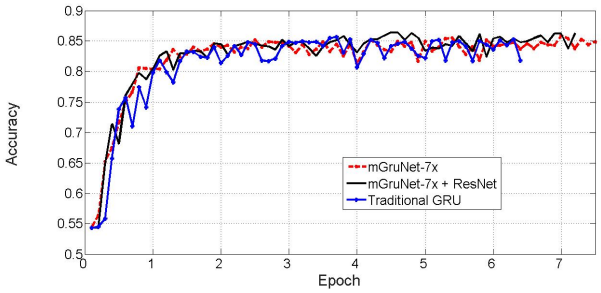


Fig. 17. Analytical training process with the 3-layer log-tree residual RNN topology.

likely to sample at a low rate and do not require a clock rate over hundreds of megahertz. The designed prototype also support a low power mode. Scaling to 20MHz and 0.8V, the overall power consumption is measured at 6.6mW, whereas the energy efficiency is 2.36 TOPS/W.

Targeting for an AI-embedded standalone micro-controller, complete peripherals, including flash memory, GPIO, UART, and JTAG, have developed with the prototype. Miscellaneous devices, such as push-buttons, crystal oscillators and power management ICs, are assembled on the board as well, demonstrated in Fig. 15. Two cases for **OCEAN** applications have been demonstrated to measure the chip performance.

A. A Single-Layer RNN for MNIST

Although designed for sequential modeling, the OCEAN processor also supports non-sequential AI modeling such as image processing. The OCEAN processor has been tested on MNIST, a well-known hand-written digit database. Each input image in MNIST has a pixel size of 28×28. To adapt the size, a single layer RNN of 32 input features and output features, namely $M = 32, N = 32$ and a fully connected layer are implemented. Before being loaded to the chip, all images are enlarged to 32×32 through zero-padding. The total time step is 32 which equals to the image column number. Considering that each accelerator is a 4-cell GRU, the 32 output features are distributed on the 8 accelerators. The overall latency is dominated by writing input feature vectors onto the FIFOs, consuming $32 \times 8 \times 32/2$ cycles. The factor of 2 is due to the ratio between the 32-bit core and 16-bit accelerators. Together with the weight DMA time, the overall RNN latency is 18.4μs and thus achieving the throughput of 54.2K frame per second. The inference accuracy on the prototype is 91.5%, which is 4.3% loss compared with the GPU-based floating point implementation.

B. Log-Tree Residual RNN for Sequential Modeling

With the RISC core, the proposed accelerators are capable of assembling flexible multi-layer deep RNNs by C programming. For example, a log-tree based multi-layer RNN network is commonly used in natural language processing applications [24]. However, DNNs are commonly confronted the issue of low convergence speed. To accelerate the convergence of multi-layer log-tree GRU networks, a linear projection matrix (LPM) was introduced in deep residual networks (ResNets) [25]. Assuming that the original DNN function is $\mathcal{F}(\vec{x})$ and LPM is a bypass FCN connecting directly from the input to the output, then the new output function $\mathcal{F}'(\vec{x})$ can be represented as

$$\mathcal{F}'(\vec{x}) = \mathcal{F}(\vec{x}) + \mathbf{LPM} \cdot \vec{x} \tag{11}$$

Fig. 16 illusltrates a three-layer log-tree residual network with seven 4-cell GRU cores. Fig. 17 demonstrates the network performance on a well-known IMDB semantic-classification data set in the NLP field [27]. The training process is completed on a GPU server by the typical analytical gradient computing. Achieving a peak accuracy of 86.4%, the log-tree network outperforms the dense 16-cell ($M = 16, N = 16$) GRU network in terms of both accuracy and convergence speed.

The maximum input feature process consumes 4.7μs with an 80-sample sequence length at a 400MHz clock rate.

Figure 18 demonstrates the on-chip incremental learning effectiveness of the OCEAN prototype with the log-tree residual network on several common data sets including IMDB sentiment classification, Amazon electronic product review [29], and EEG-based seizure detection [30]. The dataset of IMDB contains 25,000 movies reviews labeled by sentiment (positive/negative). Amazon electronic product review dataset consists of around 1 million electronic product reviews from amazon, which can also be used for sentiment classification. EEG dataset consists of thousands of electroencephalographic

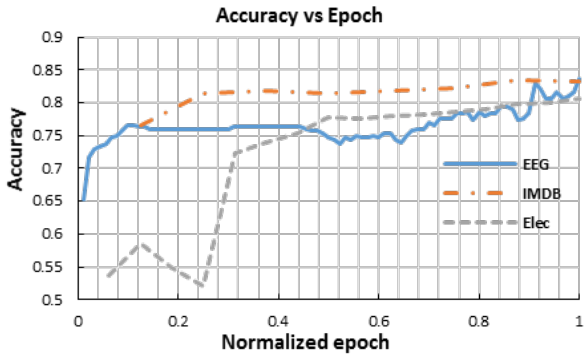


Fig. 18. On-chip incremental learning accuracy convergence.

TABLE III
PERFORMANCE SUMMARY AND COMPARISON

	2017 A-SSCC Lee [33]	2017 JSSC Moons [5]	**This Work**
Technology	65nm	40nm	65nm
ML Algorithm	RNN/FCN	CNN	GRU-RNN
Area (mm^2)	1.3 x 1.4	1.2 x 2	2.9 x 3.5
On-Chip Core	No	RISC	RISC
Peak Clock	200 MHz	204 MHz	400MHz
On-Chip Mem	10KB	144 KB	64 KB
Power	21mW	76mW	155.8mW
On-Chip Training	No	No	**Yes**
Gate #	/	1.6M	4.91M
Throughput GOPS	/	52.8	311.6
Energy Eff. TOPS/W	1.1	1.6	2.0
MNIST Classf. Throughput	/	13.4Kfps	54.2Kfps

recording waves from epilepsy patients, which can be used for epileptic seizure detection. OCEAN achieved greater than 80% accuracy with local on-chip training for all the data sets.

The performance of the OCEAN processor is summarized and compared with the state of art in Table III. The proposed prototype, to the authors' best knowledge, is the first RNN-specific silicon design achieving both effective and efficient inference and on-chip learning on edge devices. It achieved energy efficiency of 2 TOPS/W for the INT16 data format.

VII. Conclusion

This paper presented OCEAN, a deep learning processor capable of RNN inference and on-chip incremental learning. The OCEAN processor has an RISC core and eight optimized gated recurrent unit RNN accelerators enhanced with gradient computation support. On-chip incremental learning is achieved using semi-analytic gradient computation with inference accelerators. The OCEAN processor prototype has been designed and fabricated in 65nm CMOS. The prototype achieved a clock rate of 400MHz, dissipating 155.8mW. The measurement results have shown the efficiency and effectiveness of OCEAN in both inference and on-chip learning for practical edge-side AI applications.

Acknowledgments

Dr. Li Deng formally with Microsoft inspired us to initiate this research especially GRU custom silicon acceleration. The authors thank Rui Ma for his work on OCEAN RTL development, Prof. Peiyong Zhang of Zhejiang University on OCEAN phyical implementation, Meng Duan, Liang Wang, and Lu Liu on OCEAN test case preparation and demonstration, and Profs. Hao Min and Xiaolan Yan of Fudan University for their support. C-SKY Inc. provided the RISC core CK-803 and application support. The research was supported in part by the Shanghai Science and Technology Innovative Program under Grant 16JC1420300.

References

[1] Y. LeCun, B. Yoshua, and H. Geoffrey, "Deep learning," *Nature*, vol. 521, No. 7553, pp. 436-444, May 2015.
[2] J. Sim, *et al.*, "1.42TOPS/W deep convolutional neural network recognition processor for intelligent IoE systems," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2016, pp. 264-266.
[3] P. Knag, *et al.*, "A sparse coding neural network ASIC with on-chip learning for feature extraction and encoding," *IEEE J. Solid-State Circuits*, vol. 50, no. 4, pp. 1070-1079, Apr. 2015.
[4] Y. Chen, *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2016, pp. 262-263.
[5] B. Moons and M. Verhelst, "An energy-efficient precision-scalable convNet processor in 40-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 52, no. 4, pp. 903-914, Apr. 2017.
[6] G. Desoli, *et al.*, "A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28nm for intelligent embedded systems," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers,* Feb. 2017, pp. 238-239.
[7] D. Shin, *et al.*, "DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers,* Feb. 2017, pp. 240-241.
[8] S. Yin, *et al.*, "A 1.06-to-5.09 TOPS/W reconfigurable hybrid-neural-network processor for deep learning applications," in *Symp. VLSI Circuits Dig Tech. Papers*, Jun. 2017, pp. 188-189.
[9] C. Chen, *et al.*, "OCEAN: An on-chip incremental-learning enhanced processor with gated recurrent neural network accelerators," in *Proc. of Euro. Solid-State Circuits Conf. (ESSCIRC),* Sept. 2017, pp. 345-348.
[10] A. Graves, A. r. Mohamed and G. Hinton, "Speech recognition with deep recurrent neural networks," in *IEEE International Conference on Acoustics, Speech and Signal Processing,*, May 2013, pp. 6645-6649.
[11] S. Chetlur, *et al.*, "Cudnn: Efficient primitives for deep learning," *arXiv*: 1410.0759, 2014.
[12] S. Park, *et al.*, "A 1.93TOPS/W scalable deep learning/inference processor with tetra-parallel MIMD architecture for big-data applications," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers,* Feb. 2015, pp. 80-81.
[13] Y. Bengio, N. Lonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv*: 1308.3432, 2013.
[14] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits: A Design Perspective*, 2nd Edition, Pearson.
[15] A. Parashar, *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. of IEEE/ACM Int. Sympo. on Computer Architecture (ISCA)*, Jun. 2017, pp. 27-40.
[16] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *arXiv*: 1606.04838, 2016.
[17] M. D. Zeiler, "ADADELTA: an adaptive learning rate method," *arXiv*: 1212.5701, 2012.
[18] Y. Bengio, P. Simard and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. on Neural Networks*, vol. 5, no. 2, pp. 157-166, Mar. 1994.
[19] S. Hochreiter and J. Schmidhuber, "Long short term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, 1997.
[20] J. Chung, *et al.*, "Empirical evaluation of gated recurrent neural networks on sequence modeling," in *Proc. of the 32nd Int. Conf. Machine Learning (ICML)*, 2015, pp. 2342-2350.
[21] R. Polikar, *et al.*, "Learn++: an incremental learning algorithm for supervised neural networks," *IEEE Trans. on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 31, no. 4, pp. 497-508, Nov. 2001.
[22] P. Nilsson, *et al.*, "Hardware implementation of the exponential function using Taylor series," in *Proc. of the 32nd Norchip Conf.*, 2014, pp. 1-4.

[23] C. H. Tsai, *et al.*, "Hardware-efficient sigmoid function with adjustable precision for a neural network system," *IEEE Trans. on Circuits and Systems II: Express Briefs*, vol. 62, no. 11, pp. 1073-1077, Nov. 2015.
[24] K.-S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv*: 1503.00075, 2015.
[25] K. He, *et al.*, "Deep residual learning for image recognition," in *The IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770-778.
[26] Y. LeCun, *et al.*, "Gradient-based learning applied to document recognition," *Proc. of the IEEE*, vol. 86, no. 11, pp.2278-2324, Nov 1998.
[27] A. Maas, *et al.*, "Learning word vectors for sentiment analysis," in *The 49th Annu. Meeting of the Association for Computational Linguistics (ACL)*, 2011, pp. 142-150.
[28] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms," *arXiv*: 1708.07747, 2017.
[29] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in ACM Recommender Systems Conference, Oct. 2013.
[30] R. G. Andrzejak, *et al.*, "Indications of nonlinear deterministic and finite dimensional structures in time series of brain electrical activity: Dependence on recording region and brain state," *Phys. Rev. E*, vol. 64, no. 6, pp. (061907)1-8, 2001.
[31] P. Ouyang, S. Yin, and S. Wei, "A fast and power efficient architecture to parallelize LSTM based RNN for cognitive intelligence applications," in *Proc. of the 54th Annu. Design Automation Conference (DAC)*, Jun. 2017, pp. 63:1-6.
[32] Z. Wang, J. Lin, and Z. Wang, "Accelerating recurrent neural networks: a memory-efficient approach," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2763-2775, Oct. 2017.
[33] J. Lee, D. Shin, and H.-J. Yoo "A 21mW low-power recurrent neural network accelerator with quantization tables for embedded deep learning applications", in *IEEE Asian Solid-State Circuits Conf. (A-SSCC)*, Nov. 2017.

Chixiao Chen (S'10-M'17) received the B.S. and Ph.D. degrees in Microelectronics from Fudan University, Shanghai, China in 2010 and 2015, respectively. He was an exchange student in the University of California, Davis during 2008 to 2009. In 2015, he worked at Calterah Inc. as an analog/mixed signal circuit design engineer. Since 2016, he has been with the Department of Electrical Engineering, University of Washington, Seattle as a post-doctoral research associate.

His research interest includes mixed signal integrated circuit design and intelligent hardware systems. He received an Outstanding Ph.D Research Support Award from Fudan University and an IEEE Solid-State STGA Award from ISSCC 2014.

Hongwei Ding received the B.S degree in Microelectronics from Fudan University, Shanghai, China in 2016. He was a summer exchange student in Hong Kong University and the Hong Kong University of Science and Technology during 2014 and 2015, respectively. He is pursuing the Master degree in Electrical Engineering at the University of Washington. His research interest lies in SoC, VLSI design and deep learning.

Huwan Peng received the B.S. degree in Microelectronics from Shanghai Jiao Tong University, Shanghai, China in 2016. Now he is pursuing the Ph.D. degree in Electrical Engineering at the University of Washington. His research interest includes VLSI, computer architecture, and deep learning.

Haozhe Zhu received the B.S. degree in Microelectronics from Fudan University, Shanghai, China in 2017. Now he is pursuing the Master degree in Fudan University.

His research interest includes VLSI system design and computer architecture.

Yu Wang (M'16) received the B.S. and Ph.D degrees in Microelectronics from Fudan University, Shanghai, China, in 2010 and 2016, respectively. He was selected as a joint training Ph.D candidate at the University of Washington by the China Scholarship Council between 2013 to 2014. He has been a research associate in Fudan University, Shanghai, China since 2016. His research interest is in energy-efficient integrated circuit design for power management and intelligent computing.

C.-J. Richard Shi (M'91-SM'99-F'06) has been a Professor in Electrical Engineering since 2004 with the University of Washington, Seattle, where he joined in 1998. He received a prestigious Doctoral Prize from the Natural Science and Engineering Research Council (NSERC) of Canada and a Governor-General's Silver Medal in 1995 for his PhD Dissertation in computer science.

His current research interest includes energy-efficient circuit and system design for sensing, computing, learning and communication. Since 2005, he has directed several sponsored reseearch projects in the area of ADC, PLL, SerDes and LDPC design. Previously, he worked in the area of computer-aided design of mixed-signal integrated circuits, in which for his contribution he was elevated to a Fellow of IEEE in 2005. He received several awards for his research including Donald O. Pederson Best IEEE Transactions on CAD Paper Award, Best Paper Awards from the IEEE/ACM Design Automation Conference, the IEEE VLSI Test Symposium, and the SRC Technical Conference, and an NSF CAREER Award. He has served ten years as an Associate Editor of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, twice as an Associate Editor, once as a Guest Editor, of the IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing, as well as Transactions Briefs.