An Application Specific Processor Architecture with 3D Integration for Recurrent Neural Networks

Sumon Dey, Paul D. Franzon
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695, USA
{sdey, paulf}@ncsu.edu

***Abstract*—Deep learning using recurrent neural networks has broadened the horizon of artificial intelligence. It can process a massive amount of multimodal natural data (video, audio) and learn useful join representations in various applications. However, implementation of recurrent neural networks in hardware and learn representations requires high throughput and memory bandwidth of that hardware platform. This work offers a 3D hardware architecture with application-specific instruction set processor, 3D-stacked memory, and sized on-chip memory for training and inference of recurrent neural networks. It also implemented a set of short instructions after analyzing different complex, time-consuming, special operations into high-level function blocks. The accelerator also performed state-of-the-art mixed precision training using custom instructions. A high-level programming environment is developed to generate Very Long Instruction Word (VLIW) instructions for this accelerator and processed a popular and successful variant of the recurrent neural network. At 28nm, this work achieved 8.5× processing speedup, 47.5× energy efficiency per sequence, and 2.71× reduction in silicon area against a GPU.**

***Index Terms*—deep learning, accelerator, LSTM, ASIP.**

I. Introduction

Machine learning algorithms enjoyed enormous success in many modern applications ranging from healthcare to e-commerce to social network, etc [1]. The most common form of machine learning is deep learning, which can process natural data. In recent years, deep learning algorithms have outperformed most other machine learning algorithms especially in pattern-recognition, natural language processing, and in navigation (self-driving cars) [1]. The automated translation (English-to-French, English-to-German) system used the deep Long short-term memory (LSTM) and reduced the translation error by 60% compared to the Google's phrase-based system [2]. However, implementation of such deep neural networks rely on high throughput and memory intensive hardware architecture to deal with massive amounts of computation and data.

Performance is the most significant requirement of the hardware architecture to implement different variants of RNN. The architecture community has mainly focused on CNN, with little attention to RNNs or MLPs, where CNN only process tiny portion of the workload in the complete system [3]. In addition, most of the hardware research of RNN are focused on inference engine where training is still heavily involved with multiple GPUs. One of the primary motivations of this ASIP accelerator is to fill that research gap and implement a flexible hardware accelerator for training as well as inference. The training of RNN runs on multiple GPUs and takes more than a week and a single forward propagation of training requires an estimated 2 TFOPS and 12 Tb/s memory bandwidth [2]. The parameters of RNN are the floating-point dense matrices and it is not possible to store entire parameters in on-chip memory. As a result, computations mostly rely on data from off-chip memory and off-chip memory bandwidth play a crucial role in designing hardware.

This work was supported in part by the U.S. Defense Advanced Research Projects Agency (DARPA) under contract of CHIPS project.

Flexibility is another essential requirement of the hardware architecture for mapping variants of RNN. The data flow for variants of deep RNN are different and an application specific RNN shows better performance accuracy compared to generic RNN networks. Thus, a new hardware solution is required to improve training time, react in a faster manner to real-life inputs, and keep up with the change of algorithms and network variants. Flexibility of hardware is also required to map different dimension networks into that hardware in a different fashion (model parallelism, data parallelism, etc.) and achieve the best performance [4].

The next critical challenges of RNN hardware is the area and power. As discussed, hardware requires a massive amount of off-chip storage and cell area of off-chip memory and integration (2D or 3D) plays a vital role in the overall area. The half precision floating-point arithmetic is also area and power efficient (14.45× and 7.6×, respectively) and significantly decreases the total area budget [5]. Unfortunately, half precision training loses the training accuracy (convergence) for some variants of RNNs. However, the mixed-precision training, combination of single and half precision, can maintain training accuracy as good as single precision in hardware and can reduce the area of silicon [6]. The data access pattern of RNN during training is another critical challenge to design hardware. The training of RNN has a pattern of irregular data access from off-chip memory instead of contiguous behaviors. The irregular pattern of memory access makes DRAM much harder to operate on burst mode and maintain the highest DRAM bandwidth during training. So, mapping the network parameters to a DRAM and associated functionality in the hardware could increase the memory bandwidth and accelerate the training of RNN.

These are the motivational factors which led to the design of an application-specific instruction set processor (ASIP) for RNN targeting both training and inference. The contributions on this work include:

- A 3D architecture with an emerging 3D-stacked memory to increase the memory bandwidth and a sized on-chip memory to enhance data locality for accelerating learning and inference.
- A set of short instructions for different complex, time-consuming, special operations into high-level function blocks during training and inference of RNNs.
- A state-of-the-art training technique, mixed-precision training that consists of half-precision multiplier followed by the single-precision adder, to improve hardware performance maintaining the training accuracy.
- A look-up table (LUT) technique, a combination of on-chip memory based LUT and fast numerical method, and associated high-level instructions to improve the performance of hardware.
- A high-level programming environment to generate Very Long Instruction Word (VLIW) instructions for this ASIP architecture due to process various RNNs.

The rest of the paper is organized as follows. Section II discusses the related works of RNN accelerators. In section III, the architecture of proposed accelerator is presented. Section IV discusses the methodology for estimating power, speed, and area of the proposed architecture. Results and comparisons with a GPU in Section V show the acceleration and power efficiency for different batch modes. Section VI provides a conclusion.

II. Related Works

Field-Programmable Gate Arrays (FPGAs) are often used to take advantage of the programmability and rapid-prototyping capabilities of an LSTM network. FPGAs are mostly used to implement the preprocessed LSTM networks for inference mode only [7]. A relatively small size LSTM network without classification layer was implemented in [8]. In [9], different levels of parallelism and approximated activation functions were used to implement LSTM network in FPGA and performed training and inference. The performance (area, speed) was much better than that of CPUs. However, the accuracy of trained model decreased and the performance (speed) for FPGA implementation for the large scale network was worse than for GPU. A preprocessed and quantized model of LSTM was stored in on-chip memory and proposed an FPGA-based low power speech processing system in [10].

A novel compression method of LSTM is proposed in [11] to reduce memory use up to 95%. The circulant matrix-vector multiplication, different quantize methods, and an approximation of the non-linear activation function were used to implement the compressed LSTM on hardware. This work achieved significant performance improvement with minimum accuracy loss. However, the retrain process used in offline to improve the accuracy of this compressed LSTM network. The state-of-the-art ASIC acceleration of three popular neural networks (MLP, CNN, and LSTM) was in Google's Tensor Processing Unit (TPU) [3]. The TPU was implemented to accelerate Google's datacenter computation demands. The TPU was optimized to reuse weights across a large batch of independent input during inference only. This work also presented important findings of computations and resources required for three popular DNNs. CNNs are computation bound with only 5% of the total workload, and MLPs and LSTMs are the memory bound. EIE [12] proposed an energy efficient solution to accelerate sparse matrix-vector multiplication and was specialized for data sharing technique for compressed CNN and LSTM. This solution was designed targeting only inference of deep CNN (used weight sharing and dynamic Sparsity of activation).

A domain-specific Instruction Set Architecture (ISA) was presented in Cambricon [13] that target ten different neural networks. The load-store based this architecture has 64 general purpose registers and scratchpad memory instead of vector register files. The accelerator benchmarked for a small LSTM network (training and test) against GPU and on average it achieved 3.09× speedup. The integration and exploration of high bandwidth memory (3D stacking and non-volatile memory), and processing a large-scale network were left for future work.

The FPGA and ASIC implementations were based on quantized, preprocessed, and/or compressed deep neural networks for inference. The compressed, fixed-point networks were generated for inference hardware. The Cambricon is hardware capable of both training and inference. This work presents an instruction based 3D architecture with stacks of emerging memories for both training and inference. It is different than [13] in the sense that it enhances memory bandwidth using 3D memory, issues up to five instructions (VLIW), uses mixed-precision computation (required for training) and can process a large DNN on natural datasets (needed high storage).

III. Hardware Implementation Details

The details of proposed ASIP accelerator for training and inference of RNNs are presented in this section. This section also discussed the features of ASIP architecture to address various challenges.

A. System Overview

A schematic representation of the overall proposed architecture is shown in Fig. 1 and Fig. 2. It is a 3D architecture to accelerate the training and inference of LSTM, MLP, and potentially other DNNs. The entire system can be divided into two different sub-systems: 1) the accelerator (multiple parallel PEs); and 2) the 3D memories are connected through the on-chip crossbar network. The host with CPU and host memory communicates to the accelerator and 3D memory using host interface of this accelerator. The overall concept of this design is that the host will offload the entire computations or the hotspot of LSTM computations to this accelerator. The host transfers the instructions, samples of input, weight

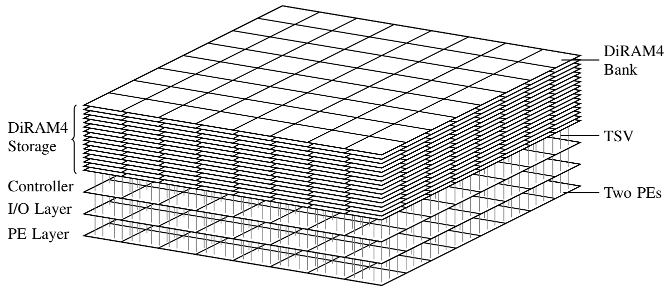

Fig. 1. The overall architecture of proposed system

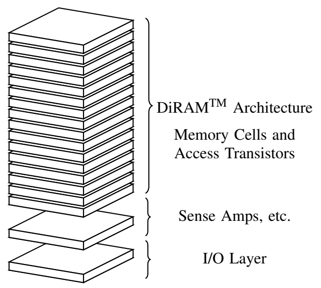

Fig. 2. A single bank of DiRAM4 3D memory

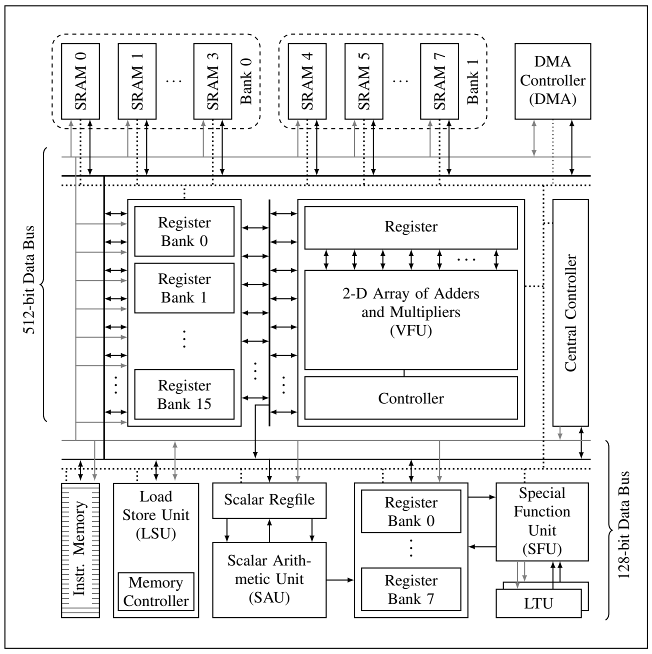

Fig. 3. The overall architecture of a PE

parameters, and hyperparameters to the 3D memory and passes the initial configurations and memory allocations to this accelerator at the beginning of the computation. The high bandwidth Tezzaron DiRAM4 is used as 3D memory in this architecture. A C++ program is written to analyze the LSTM network and generate a sequence of VLIW instructions (for memory transaction and also computation) of this accelerator. The accelerator performs forward propagation, and weight update stages for an LSTM network using different memory accesses operations to bring data inside PE from DiRAM4, load-store operations for data movement inside a PE, computations for different matrix-matrix, matrix-vector, special function operations (Sigmoid, Exponential, and Tanh) and at the end, store the output or intermediate results to the DiRAM4 using DMA operations. In case of training, the backward propagation is performed using a smart DMA controller to bring weight or intermediate parameters in the reverse order and perform different operations (load-store, matrix-matrix, special function operations, and others) inside PEs accordingly. The operations involved in three different stages (forward propagation, backward propagation, and mini-batch GD) are performed using custom instructions designed for this accelerator.

B. DiRAM4 3D Memory

The memory system has a significant impact on the overall performance of this accelerator. A banked and multi-ported 3D memory system (DiRAM4) are used with multiple identical PEs to perform the parallel computation for training and inference. It is a heterogeneous 3D stacked memory system with 18 dies (16 layers contain memories, 1 layer contain sense amps, drivers, decoders, and 1 layer contain I/O buffer and other support circuits), where it has 64 DiRAM4 banks associated with 64 ports [14]. The storage of each DiRAM4 bank is 1 Gb and the total 64 Gb storage of this memory system. The storage of individual DiRAM4 bank is arranged in 64 banks in which 4096 pages per bank and 4096 bits per page reduce bank conflict significantly. The 20-bit unidirectional address and control bus, a 32-bit unidirectional input data bus, and a 32-bit unidirectional output bus can transfer data to/from host or PEs through the on-chip crossbar network. The DiRAM4 has Dual Single Data Rate (DSDR) interface on 64 external ports with two channels to the users. Each channel operates based on its clock which is derived from the port clock (90 degrees out of phase from each other). So, the data per channel is single data rate, but the net data rate is two times the single channel (DSDR). Also, DiRAM4 can be configured either burst of 2 version (64-bit quantities) or burst of 8 version (256-bit quantities). Each port operates on a 1 GHz clock, and maximum achievable bandwidth can be as much as 8 Tb/s (read bandwidth 4 Tb/s and write bandwidth 4 Tb/s) from 64 disparate DiRAM4 banks. The proposed accelerators have 128 identical PEs connected to 64 DiRAM4 banks. The page open and access latency of DiRAM are 22ns and 5ns, respectively.

C. Processing Element

There are 128 PEs is this architecture runs independently i.e. each PE perform computation a subset of entire processing. For example, an LSTM network (two hidden layers with 1024 units each and an output layer with 16348 output units) mapped to this architecture as model parallelism fashion where an individual PE processes 16 units of hidden layer and 128 units of the output layer. The PE reads/writes from DiRAM4 complying with instructions as well as memory allocations from the host. A single PE has a five-stage pipeline ($fetch$, $issue$, $read$, $execute$, $writeback$) that can fetch four instructions (each instruction 32-bit wide) at every cycle. In addition, PE has five different lanes which allow it to execute five different types of instruction (direct memory access, load-store, vector, scalar, and special function) at the same time. The flow control mechanism of PE allows the back-to-back issue of independent instructions in each lane. A single PE contains a direct memory access (DMA), a load-store unit (LSU), four vector functional units (VFUs), a scalar arithmetic unit (SAU), four special functional units (SFUs), and memories (SRAM and register files). The overall architecture of a PE is shown in Fig. 3 and different key features are discussed in the subsequent section.

1) Direct Memory Access: A fully programmable DMA is designed to transfer bulk data between DiRAM4 and PEs in different fashions (serial-in-parallel-out, parallel-in-serial-out, 16-bit IEEE 754 to 32-bit IEEE 754 conversion and vice versa, and also scatter-gather). A set of instructions has been implemented in DMA to access data efficiently from DiRAM4 for different computational steps of training and inference. Instructions of DMA communicate its status with instructions of other execution lanes through the central control unit and the dependent instructions are scheduled accordingly. The total on-chip SRAM (2 Mb) of a PE is decomposed into two banks (each 2048×512), where DMA can access one of the banks for read/write operation from/to the DiRAM4 via crossbar network. The DMA is bidirectional and can read/write 64-bits of a data word from/to the DiRAM4 in every clock cycle. During mixed precision training, each parameter can be either 16-bit or 32-bit where 64-bit data between DMA and DiRAM4 can have either 4 or 2 parameters. In addition, the interface between DMA from/to SRAM bank is 512-bit wide which requires DMA to have data gathering/scatter-gather mechanism before write to the SRAM banks.

2) On-chip Memories: There are three different register files (two vector and one scalar) with multiple ports and eight small on-chip SRAM (each 2048×128) available in a single PE. The eight SRAM banks group into two SRAM banks (four small SRAM banks in each group) with data bus width of 512-bit. The overall concept behind two SRAM banks is to perform computation on data of one SRAM bank and pre-fetch data for next set of computations into another SRAM bank simultaneously.

The PE accommodates one multi-ported vector register file ($vreg$) size of 64 Kb and one multi-ported hybrid register file ($hreg$) of size 2 Kb. In addition, it has a multi-ported scalar register file ($sreg$) size of 4 Kb. The multiple register files with multiple ports are designed to support instruction level parallelism within five lanes of this architecture. For instance, the $vreg$ register file has three read ports and two write ports. The first read and write port are dedicated to the load-store unit and support continuous 512-bit data streaming from SRAM to $vreg$. The other two read ports are dedicated to the vector function unit (VFU) for two operands of vector operations. In addition, the write port is assigned to write vector outputs (from VFU) to the $vreg$.

3) Load-store Unit: The load-store unit (LSU) of this architecture supports different load-store instructions and perform data movements in different fashions among different on-chip memories (SRAM, $vreg$, $hreg$, and $sreg$). The LSU can load/store 512-bit data from/to SRAM to/from $vreg$ through the 512-bit data bus. It can also load/store 128-bit data from/to SRAM to/from $hreg$. In addition to the vector load/store, LSU can also read 16-bit data word from SRAM and fan it out to 512-bit or 128-bit before loading it to the $vreg$ and $hreg$, respectively. This type of load (scalar read followed by vector load) is necessary to load different hyperparameters (learning rate, momentum, etc.) of the network before performing scalar-vector operations in VFU or SFU. Moreover, the mixed precision training required to scale the loss function, which is also a scalar-vector multiplication.

4) Vector Function Unit: There are four VFUs in a single PE, where each VFU has sixteen IEEE 754 compliant half precision multiples and seventeen IEEE 754 compliant full precision adders. A Wallace tree multiplier using 3:2 compression technique was used during designing the 16-bit floating-point multiplier. The multiplier and adder of this design are fully pipelined and throughput in every clock cycle. The precision of adder is higher to support the mixed precision training as per [6]. The VFU has 4-bit operation code, two vector operands read from two different buffer connected to the vector register file, and the output (scalar or vector output based on different operation codes). The operands are continuously streamed from the $vreg$ and make available to the VFU during read stage of the pipeline. Based on different operation code during the execute stage, the output of VFU can be scalar or vector and written output to the $sreg$ or $vreg$ register file, respectively. A single instruction of VFU performs on vector operands of size 64 with a required precision conversion. For example, the MAC operation code mainly performs the dot product of 64 elements in half-precision multipliers and do the necessary conversion (half-precision to single-precision) before performing the reduction in single-precision adders. The special instructions have also added to support the loss scaling and weight update steps in full-precision required for the mixed precision training. The VFU has a 3-bit address and 4-bit operation code to support eight different operations for various computation steps of training and inference. The 4-bit operation code of VFU offers the possibility of future extension for other vector operations.

5) Scalar Arithmetic Unit: A single Scalar Arithmetic Unit (SAU) in each PE performs different scalar operations as well as scalar-to-vector conversion and writes the vector to the $hreg$ register file. The SAU consists of a single precision adder, a half precision multiplier, a half precision multiplier-based divider, and a buffer. The scalar outputs from VFU are usually passed to the $sreg$ for different scalar operations and/or generates a vector output of length 4 (64-bit) and write that vector to the $hreg$ register file. The SAU is also used to compute different scaling parameters (scaling factor or shanking factor) during the backward propagation of training.

TABLE I
THE STATUS OF FUNCTIONAL UNIT

Functional Unit	Opcode	Status
DMA	0100	Doing scatter-gather operation
LSU	0001	Loading data from SRAM to vreg
VFU	0001	Performing Mac operation
SAU	0010	Doing division operation
SFU	0011	Doing tanh function

TABLE II
THE STATUS OF SRAM BANK

SRAM	Functional Unit	Status
Bank 0	DMA	Scatter-gather on SRAM bank 0
Bank 1	LSU	Load data from Bank 1 to $vreg$

For example, the scaling factor is defined based on the scalar operation (division) between the maximum normalize gradient (hyperparameter) and the current normalize gradient to scale the parameter gradients of the network before weight update rule.

6) Special Function Unit: There are four special function units (SFUs) in a PE to support computation of various non-linear activation functions as well as other vector operations (addition/subtraction, multiplication). The input vector size of SFU is 4 (unlike 64 of the VFU), and it reads/writes operands/output from/to the hreg register file. The frequently used non-linear activation functions in RNNs are parameterized sigmoid, exponential, and hyperbolic tangent. The accuracy and efficiency of the non-linear functions have a significant role in the hardware performance as well as the accuracy of learning and inference for an LSTM network. The fixed-point implementations (based on Piecewise Linear (PWL), CORDIC and Taylor series expansion) are the most common approaches to implement the non-linear functions, which always suffer small to medium accuracy loose for different input range. To maintain computation in IEEE 754 standard and efficiency of the algorithm, a combination of the look-up table (LUT) and the numerical method is implemented in hardware to improve hardware performance and maintain the learning/inference accuracy [15]. The SFU also has its 3-bit address and seven instructions to perform different operations. A single instruction is assigned against each non-linear function to accelerate the non-linear function and at the same time reduce the number of instructions.

7) Central Controller: Multi-ported register files are designed in central controller to monitor the status of hardware resources and the data dependencies due to eliminating the structural and data hazards accordingly. During $issue$ stage of pipeline, the five instructions for five different lanes go to the central controller and simultaneously check the register file (multi-ported register file) to ensure the execution unit and/or destination register (applicable only for VFU, SAU, and SFU) of these 5 instructions are available (avoid structural and write-after-write (WAW) hazard). The successful $issue$ of one/more instructions also updates the register files of the central controller. During $read$ stage, the five instructions check the status of their operands simultaneously from the register file database and eliminate read-after-write (RAW) hazard. As soon as the execution unit finishes the execution of an instruction, the central control updates its database (free the function unit) and inform waiting instructions that the execution unit is ready to execute the next instruction. In $writeback$ stage, instructions update the status of its destination registers in the central control unit. The central control unit then informs the instructions waiting for the same destination registers (avoid WAR hazard). The unique concept of the central controller is to design multi-ported register files in a way to support multiple reads and writes during $issue$, $read$, $execute$, $writeback$ stages and serve the maximum degree of parallelism of the pipeline stages. A register-based central control named scoreboarding also used in the out-of-order processor, but a scoreboard mechanism can issue only one instruction at a time where this architecture can issue a maximum of five instructions at a time. Table I, Table II, and Table III represents an example of the register file content in the central controller.

TABLE III
THE STATUS OF REGISTERS IN REGISTER FILES

Name	Address	Status
vreg	0	Busy
vreg	1	Available
vreg	2	Available
vreg	3	Busy
⋮	⋮	⋮
hreg	0	Available
hreg	1	Busy
hreg	2	Busy
hreg	3	Available
⋮	⋮	⋮
sreg	0	Busy
sreg	1	Busy
sreg	2	Available
sreg	3	Available
⋮	⋮	⋮

D. Instruction

The five stages pipeline-based this architecture has five different execution lanes (DMA, LSU, VFU, SFU, SAU) and five different addresses assigned to these execution lanes. There are also five groups of different instructions corresponding to these execution lanes. An instruction format and an overview of sample instructions of this architecture are presented in Table IV and Table V, respectively. In addition, each execution lane supports "no operation" for pipeline bubble and introduces delay inside the pipeline. The "no operation" is required for C++ compiler to insert a delay in the execution lane for maintaining the register-based flow control among all lanes and avoid different hazards (structural and data hazard). For instance,

TABLE IV
AN INSTRUCTION FORMAT OF LANE

31–29	28–25	24–0
Lane Address	Opcode	Addresses and others

TABLE V
AN OVERVIEW OF DIFFERENT INSTRUCTIONS

Lane Type	Operation	Description
DMA	Config	Program DMA for bulk transfer
DMA	SG	Read DiRAM and gather to SRAM
VFU	VMAC	Perform multiply–accumulate
VFU	VADD	Perform element-wise addition
LSU	VLDR	Load SRAM to the $vreg$
LSU	VLSH	Load SRAM to the $hreg$
SAU	SADD	Perform scaler addition
SAU	SDIV	Perform scaler division
SFU	VSIG	Perform Sigmoid calculation
SFU	VTAN	Perform Tanh calculation

the VFU execution lane is fully pipelined and supports eight different data paths for eight different instructions. The central controller has four cycle delays in updating its register during issuing an instruction. If it requires to issue a different type of VFU instructions back-to-back, then there should have at least four cycles delay to eliminate the datapath conflict (structural hazard). The compiler introduces four cycles delay between two different types of VFU instruction with the help of "no operation". However, issuing the same type of VFU instruction back-to-back does not require to insert any delay inside the pipeline. There is also a "synchronous" instruction in each execution lane to synchronize execution among different lanes. The compiler usually required to use "synchronous" instructions especially after finish computation of one LSTM layer and before start executing instructions for the next layer.

E. Data Reuse

The off-chip memory access (DiRAM4) is always expensive from latency (22 ns delay for worst case page open and 14 ns delay for worst case access) and power point of view (page open and cache read/write energies are 100 pJ and 64 pJ, respectively, at 1 GHz). Due to speed up of the overall performance, the data reuse technique has been used at various steps of learning and inference for an LSTM network. The LSTM network offers very small data reusability. However, in batch processing, the multiple examples/samples are applied to the same weight parameters and the weight reusability is proportional to the number of samples or the batch sizes. These operations (matrix-matrix or matrix-vector multiplication) are performed using a block matrix where submatrices of weight and input matrix are kept inside PE and the other submatrices are continuously streamed from the off-chip memory. The data reuse is done at various stages inside a PE. The on-chip SRAM holds the partial weight parameters and at the same time, $vreg$ can hold a segment of those parameters during batch processing. In addition, the state of the LSTM unit, the error

TABLE VI
AN EXAMPLE OF HIGH-LEVEL C++ FUNCTION

High-Level Function	Generated Instruction	Description
DLDR(0, 1, 0, 15)	22600000	Reg[0] $\Leftarrow$ 0
	22610010	Reg[1] $\Leftarrow$ 15
	22620000	Reg[2] $\Leftarrow$ 0
	22630000	Reg[3] $\Leftarrow$ 0
	22640002	Reg[3] $\Leftarrow$ 1
	22650000	Reg[4] $\Leftarrow$ 0
	22660001	Reg[5] $\Leftarrow$ 1
	22670001	Reg[6] $\Leftarrow$ 1
	24600006	Channel Request0
	26600000	Start DMA Operation
VLDR(0, 2)	a2000002	vreg[2] $\Leftarrow$ SRAM[0]
VMAC(0, 1, 0)	82000100	sreg[0] $\Leftarrow$ Mac(vreg[0], vreg[1])
SADD(0, 1, 2)	62000102	sreg[2] $\Leftarrow$ sreg[0] + sreg[1]
VTAN(0, 1)	46000001	hreg[1] $\Leftarrow$ Tanh(hreg[0])

of current timestamp (usually used in the previous timestamp during backward propagation) and intermediate activations (activations of gates, cells, etc.) are stored in on-chip SRAM and used once or multiple times in subsequent computations.

F. Compiler

There could be several ways to generate instructions of an LSTM network. In this implementation, the data flow of an entire LSTM network has been analyzed during compile time and allocated memories for efficient memory management for both training and inference. The high-level C++ functions have designed to generate the low-level VLIW instructions into a binary text file to perform off-chip memory transaction at various forms, block matrix operations, data movement, and also various special function operations. The current version of C++ implantations is designed targeting the LSTM and MLP networks. However, the implementation can be extended for other DNNs in future. The sample C++ functions and generated instructions for five different functional units are explained in Table VI.

The first high-level function (DLDR) generates a sequence of instructions to pull data out from the DiRAM4 and load to the SRAM. The configure instructions (first nine instructions) program the DMA registers and channel assignment for this transfer. Then, the start instruction transfers (16×64) 128 bytes of DiRAM4 data (between address 0 and 15) to the SRAM address 0 and 1 (SRAM bus is 512-bit wide). The VLDR function generates instructions to load 512-bit data from SRAM address 0 to the $vreg$ at address 2. The VMAC function generates an instruction to perform MAC operations on data located at address 0 and 1 of the $vreg$ and the write scalar output to the $sreg$ at 0 address. Similarly, the SADD and VTAN functions generate an instruction to perform the scalar addition and tangent hyperbolic, respectively. Based on these instructions, the next level high-level functions were also written in the compiler to generate VLIW instructions for forward propagation, backward propagation, and weight

TABLE VII
THE KEY HARDWARE RESULTS OF A PE

Name	Value
Technology	28 nm
Clock Frequency	500 MHz
Core Voltage	0.8 V
Total Area	1.54 mm^2
Total Cell Area	1.07 mm^2
On-chip SRAM Area	0.47 mm^2
On-chip Register	26 KB
On-chip SRAM	256 KB
SRAM Area/Total Area	31 %
Core Power	256.76 mW

update stages of the target LSTM network for both batch and non-batch processing.

IV. Experimental Setup

The Verilog RTL was used to model and implement a single PE and memory generator was used to generate the on-chip memory (SRAM). The behavioral model of a DiRAM4 was also implemented in Verilog and used as external 1 Gb DiRAM4 bank. The storage of parameters, PBT dataset, and data flow of the target benchmark network was analyzed during compile time and assigned to the DiRAM4 memory accordingly [16]. The high-level C++ function was called to generate instructions for processing the target network and dump binary instructions into a text file. The instructions were loaded to the instruction buffer.

The ModelSim simulation was performed to calculate the number of cycles required to process a benchmark network. The C++ environments have been developed for functional verifications of high-level function as well as the benchmark network. The synthesis of RTL was done using Synopsys DesignWare in 28nm commercial standard cell library at 500 MHz, and the overall area of a PE was calculated. The total area of this architecture was calculated based on the area of a single PE. The power of a PE was calculated based on the routed netlist after place-and-route of that PE using Synopsys ICC followed by switching activity (VCD) processed by Synopsys Primetime. In addition, the access pattern of DiRAM4 has been capture in ModelSim simulation and calculated power based on access (page open/close, cache read/write, page refresh, etc.) energy from the DiRAM4 datasheet. The spreadsheet calculation was performed to calculate the overall performance for comparing with a GPU. To profile a Tesla K40c GPU, the baseline network as well as profiling script were written in torch7 (deep learning framework) using deep learning APIs (cunn, cuDNN, cutorch) for CUDA backend acceleration. The die area and power of GPUs were taken from the datasheet of respective GPU.

V. Results

This section summarizes the area, power, and timing analysis of this architecture with Tesla K40c GPU. The performance comparison of this accelerator is evaluated using an LSTM network for language processing task (i.e., word-level prediction experiments on the Penn Tree Bank (PTB) dataset). The LSTM network has two LSTM layers (1024 units per layer) and an output layer (16384 output units). The network was unrolled for 32 time steps and evaluated training and inference performance for two different mini-batch sizes (1 and 32). The online or non-batch processing means when minibatch size is 1. The batch processing of this architecture was performed for a mini-batch size of 32.

TABLE VIII
THE DIE AREA COMPARISON

Name	Tesla K40c GPU	This Work	Improvement
Die Area (mm^2)	561†	207	2.71×

†die area from [3]

The key hardware results of a single PE is shown in Table VII. The PE is synthesized at 500 MHz in 28nm commercial technology followed by place-and-route for calculating the average power. The three on-chip register files (64 Kb for $vreg$, 2 Kb for $hreg$, and 8 Kb for $sreg$), and sequential cells (22 Kb) adds up total 26 KB on-chip register of a PE. The size of on-chip SRAM is 256 KB (31% of the total area), which is usually useful for improving data locality during batch processing. Based on the area of a single PE and considering 5% of the area for the crossbar network and other logic in the PE layer, the overall area of this architecture was estimated. The overall area comparison against a GPU's die area is shown in Table VIII. The area benefit of this architecture is 2.71×. The die area of DiRAM4 3D memory is the same as the die area of this architecture, which is essential for efficient 3D integration.

The processing time, power, and memory bandwidth requirement for batch processing are shown in Table IX. The batch inference performance refers to the resource requirements due to process 32 samples unrolled up to 32 time steps for forward propagation only. The batch training refers to the resource requirements for processing a forward propagation, a backward propagation for 32 samples unrolled up to 32 time steps, and a weight update stage. This architecture speeds up the batch processing 2.58× and 2.46× for inference and training, respectively. The high bandwidth DiRAM4 has helped this architecture to achieve approximately 8× speedup during online training and inference. The on-chip SRAM and reduced precision arithmetic of this architecture have also contributed to achieving almost 6× power benefit again that GPU. The memory bandwidth requirement for online processing of this architecture is approximately 3.6 Tb/s. The speed up, power benefits, and bandwidth requirement for online processing of this architecture are shown in Table X.

VI. Conclusion

In this work, a novel 3D architecture with an application-specific instruction set architecture have been proposed to accelerate training and inference of RNN, MLP, and poten-

TABLE IX
THE SPEED AND POWER COMPARISON FOR BATCH PROCESSING

Name	**Tesla K40c GPU**		**This Work**		**Improvement**	
Type of Processing	**Batch Inference**	**Batch Training**	**Batch Inference**	**Batch Training**	**Batch Inference**	**Batch Training**
Processing Time (ms)	126.69	398.64	49.16	162.26	2.58×	2.46×
Power (W)	235	235	36.36	37.95	6.46×	6.19×
Memory BW (GB/s)	208	208	1024	1024	4.92×	4.92×

Notes. the board power and bandwdith of GPU from datasheet

TABLE X
THE SPEED AND POWER COMPARISON FOR ONLINE PROCESSING

Name	**Tesla K20c GPU**		**This Work**		**Improvement**	
Type of Processing	**Online Inference**	**Online Training**	**Online Inference**	**Online Training**	**Online Inference**	**Online Training**
Processing Time (ms)	40.66	171.39	4.82	21.99	8.44×	7.79×
Power (W)	235	235	36.37	39.15	6.46×	6.10×
Memory BW (GB/s)	208	208	1024	1024	4.92×	4.92×

Notes. the board power and bandwdith of GPU from datasheet

tially other DNNs. An emerging 3D memory has adapted in this architecture to improve the memory bandwidth. The on-chip memory has been used to increase the data locality for batch processing, and improve power and bandwidth budget. In addition, instructions were designed carefully to reduce the number of instructions and the instruction bandwidth. Multi-ported register files have designed and implemented to support the highest degree of instruction-level parallelisms. Instructions were added to support both half-precision and mixed precision training due to maintaining the accuracy as good as the single precision. A small lookup table and numerical method were implemented in hardware to perform special function operations and improve the hardware performance. Dedicated instructions for special function operations were also incorporated in this work. The high-level C++ functions were written to generate blocks of instructions to perform forward propagation, backward propagation, and weight update stages. The overall improvement of this work was 44.47× – 147.75× for various batch modes. Future work includes adding instructions after careful evaluation of other DNNs (CNN, RBM, etc.) and benchmark accordingly. The final chip tape-out also remains to be done on as future work.

ACKNOWLEDGMENT

The author would like to thank Lee Baker, Joshua Schabel, and Weifu Li for their interesting discussions on this work.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[2] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[3] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 2017, pp. 1–12.

[4] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *arXiv preprint arXiv:1802.09941*, 2018.

[5] W. Dally, "High-performance hardware for machine learning," *NIPS Tutorial*, 2015.

[6] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[7] A. Rush, A. Sirasao, and M. Ignatowski, "Unified deep learning with cpu, gpu, and fpga technologies," Advanced Micro Devices, Tech. Rep., 2017.

[8] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on fpga," *arXiv preprint arXiv:1511.05552*, 2015.

[9] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu, "Fpga acceleration of recurrent neural network based language model," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2015, pp. 111–118.

[10] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung, "Fpga-based low-power speech recognition with recurrent neural networks," in *Signal Processing Systems (SiPS), 2016 IEEE International Workshop on*. IEEE, 2016, pp. 230–235.

[11] Z. Wang, J. Lin, and Z. Wang, "Accelerating recurrent neural networks: A memory-efficient approach," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2763–2775, 2017.

[12] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 243–254.

[13] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 393–405.

[14] T. Semiconductor. Tezzaron diram4 3d memory. [Online]. Available: https://www.tezzaron.com/applications/diram4-3d-memory/. [Accessed Oct. 9, 2018].

[15] P.-T. P. Tang, "Table-driven implementation of the exponential function in ieee floating-point arithmetic," *ACM Transactions on Mathematical Software (TOMS)*, vol. 15, no. 2, pp. 144–157, 1989.

[16] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of english: The penn treebank," *Computational linguistics*, vol. 19, no. 2, pp. 313–330, 1993.