Hardware Accelerator for Differentiable Neural Computer and Its FPGA Implementation

Akane SAITO, Yuki UMEZAKI and Makoto IWATA
Graduate School of Engineering, Kochi University of Technology,
Kami, Kochi, 782-8502 Japan

Abstract—*Along with the recent progress of deep artificial neural network (DNN) research, accelerator circuits aiming at high speed of DNN has been proposed. These are dedicated for conventional multilayer DNNs so that they are lack of efficiency to execute a newly proposed neural network such as Differentiable Neural Computer (DNC). Since the DNC is composed of a long-term short-term memory (LSTM) and an external memory, it is necessary to calculation of the LSTM and reading / writing operations high speed to the external memory.*

In this research, a pipelined accelerator is studied in order to execute a weighted sum and its non-linear activation function by a single instruction. In this accelerator, input data and weights stored in each SRAM module fetched, and them is multiplied and accumulated consecutively. The final resultant data is written back into the SRAM module. It is designed for Stratix V, Intel FPGA chip, and the performance evaluation was carried out.

Keywords: Differentiable Neural Computer, Hardware Accelerator

1. Introduction

Recent progress of artificial neural networks (ANNs) is remarkable for their recognition capability in various application fields such as speech recognition, computer vision, and natural language processing. Especially deep neural networks (DNN) attracts attention in complex pattern recognition, e.g., image or data mining applications. Those applications operates on various platforms from data centers to small IoT devices, so that low-power and high-performance accelerators has been developed in many research communities [1], [2], [3].

However, DNNs are limited in their ability to represent data structures and to store data over long timescales without interference. This issue has been recently solved by differentiable neural computer (DNC) that was developed by the researchers in Google DeepMind Technologies Limited [4]. A DNC is a neural network coupled to an external memory matrix where data structures and long sequence of data are stored from and referred to the controller networks consecutively. Actually Alex, et al. revealed that DNC can learn tasks to find relationships between nodes in a randomly generated graph and DNC can generalize these tasks to specific graphs such as transport networks and family diagram [4]. In order to realize the more general accelerators supporting DNCs, it is necessary to provide more complex neural network such as long-short term memory (LSTM) and external memory access.

Therefore, this paper discusses a high-performance and programmable hardware accelerator supporting DNCs as well as DNNs. Since primary basic operation in these ANN is a product-sum of an input vector and a weight vector and it is followed by a non-linear function, a set of basic operations must be efficiently calculated in parallel. In general, there are several parallel processing techniques for such a programmable and dedicated core; parallel processing on many cores or a single core, data or instruction parallelism, concurrent or pipelined processing, and so on.

In this paper, we focused on a dedicated single core architecture that can deal with various types of neurons, external memory access defined in DNC, and its memory access parameter calculations. The proposed architecture is designed in terms of so-called instruction pipeline in order that its instruction stream could flow smoothly, i.e., no pipeline stall occurs on it. An instruction set of the proposed architecture is defined to cover some complex instructions, such as product-sum and non-linear (sigmoid or hyperbolic tangent) function. It is also considered to be a super-scalar extension or many core extension in the future.

The reaming part of this paper is organized as follows. In the next section, specific features of the DNC is briefly discussed in terms of designing its efficient accelerator. Section 3 presents an instruction-set architecture and micro-architecture of the proposed accelerator. Section 4 shows preliminary evaluation results acquired through an FPGA implementation of the proposed accelerator. Finally, section 5 concludes the paper.

2. Computational Structure of DNC

The DNC is a neural network coupled to an external memory. Fig. 1 shows the structure of the DNC model. The neural network part called controller network plays a role of a differentiable processor and the external memory can be accessed by differentiable attention mechanisms through the read and write heads. The read heads return the read vectors $\boldsymbol{r}^1_{t-1}, ..., \boldsymbol{r}^R_{t-1}$ which is a weighted sum over the

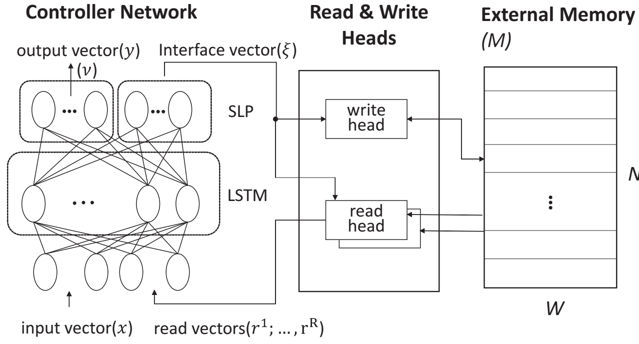

Fig. 1: An overview of DNC architecture.

memory locations with a read weighting $\boldsymbol{w}^r$ over the $N \times W$ matrix M. The write operation also uses a write weighting $\boldsymbol{w}_t^w \in \mathbb{R}^W$ with an erase vector $\boldsymbol{e}_t \in \mathbb{R}^W$ and a write vector $\boldsymbol{v}_t \in \mathbb{R}^W$. In general, any neural network can be used for the controller, but we suppose to use the deep LSTM architecture in this paper.

The basic computations of general multi-layer perceptron (MLP) ANN are the forward non-linear activation and backward learning. The forward computation is the product-sum of the input vector and the weight vector followed by the application of the nonlinear function, e.g., sigmoid, hyperbolic tangent, and so on. The backward learning is done with the gradient descent of the error in entire output. The fact that this series of operations in necessary is also applicable to the DNC, but in addition to that, the DNC needs the LSTM operation of the controller network and the read and write operations to the external memory. To compute the DNC at high speed, it is necessary to perform a series of those operations efficiently.

The DNC periodically accepts an input vector $\boldsymbol{x}_t \in \mathbb{R}^X$ and generates an output vector $\boldsymbol{y}_t \in \mathbb{R}^Y$. The controller network is structured with the deep LSTM network and the single layer perceptron. An input vector of the controller network is given by $\boldsymbol{\chi}_t = [\boldsymbol{x}_t; \boldsymbol{r}_{t-1}^1; ...; \boldsymbol{r}_{t-1}^R] \in \mathbb{R}^{X+WR}$ and an output vector of that is given by a concatenation of an intermediate output vector $\boldsymbol{\nu}_t \in \mathbb{R}^Y$ and an interface vector $\boldsymbol{\xi}_t \in \mathbb{R}^{WR+3W+5R+3}$. The LSTM architecture is defined as follows:

$$\boldsymbol{i}_t^l = \sigma(W_i^l[\boldsymbol{\chi}_t; \boldsymbol{h}_{t-1}^l; \boldsymbol{h}_t^{l-1}] + \boldsymbol{b}_i^l) \quad (1)$$
$$\boldsymbol{f}_t^l = \sigma(W_f^l[\boldsymbol{\chi}_t; \boldsymbol{h}_{t-1}^l; \boldsymbol{h}_t^{l-1}] + \boldsymbol{b}_f^l) \quad (2)$$
$$\boldsymbol{s}_t^l = \boldsymbol{f}_t^l \boldsymbol{s}_{t-1}^l + \boldsymbol{i}_t^l tanh(W_s^l[\boldsymbol{\chi}_t; \boldsymbol{h}_{t-1}^l; \boldsymbol{h}_t^{l-1}] + \boldsymbol{b}_s^l) \quad (3)$$
$$\boldsymbol{o}_t^l = \sigma(W_o^l[\boldsymbol{\chi}_t; \boldsymbol{h}_{t-1}^l; \boldsymbol{h}_t^{l-1}] + \boldsymbol{b}_o^l) \quad (4)$$
$$\boldsymbol{h}_t^l = \boldsymbol{o}_t^l tanh(\boldsymbol{s}_t^l) \quad (5)$$

where l is the layer index, $\sigma(x)$ is the sigmoid function, $\boldsymbol{h}_t^l$, $\boldsymbol{i}_t^l$, $\boldsymbol{f}_t^l$, $\boldsymbol{s}_t^l$, and $\boldsymbol{o}_t^l \in \mathbb{R}^{N_l}$ are the hidden, input gate, forget gate, state and output gate activation vectors, respectively, of layer l $(1 \leq l \leq L)$at time t. The W_* terms are learnable weight matrices and $\boldsymbol{b}_*$ are learnable bias vectors.

The intermediate output vector, the interface vector, and the output vector are respectively defined as follows:

$$\boldsymbol{\nu}_t = W_y[\boldsymbol{h}_t^1; ...; \boldsymbol{h}_t^L] \quad (6)$$
$$\boldsymbol{\xi}_t = W_\xi[\boldsymbol{h}_t^1; ...; \boldsymbol{h}_t^L] \quad (7)$$
$$\boldsymbol{y}_t = \boldsymbol{\nu}_t + W_r[\boldsymbol{r}_t^1; ...; \boldsymbol{r}_t^R] \quad (8)$$

$\boldsymbol{\nu}_t$ and $\boldsymbol{\xi}_t$ can be concatenated into a single-layer perceptron and the weight matrices W_y and W_ξ are learnt on it.

As for the external memory, the read and write heads access to the memory along with some weightings as follows:

$$\boldsymbol{r}_t^i = M_t^\top \boldsymbol{w}_t^{r,i} \quad (9)$$
$$M_t = M_{t-1} \circ (E - \boldsymbol{w}_t^w \boldsymbol{e}_t^\top) + \boldsymbol{w}_t^w \boldsymbol{v}_t^\top \quad (10)$$

where $\circ$ denotes element-wise multiplication, and E is an $N \times W$ matrix of ones. The detail computation of those weightings is defined in the original article [4].

To summarize the computational structure of the DNC at every time step, there are four major components as listed below.

- Forward computation of the controller network
- Backward learning of the controller network
- Read weightings and operations by R read heads
- Write weightings and operations by a write head.

3. Architecture of Hardware Accelerator

The hardware accelerator for the DNC discussed here is aimed to keep the smooth instruction stream within its instruction pipeline. Therefore, the processing core of the hardware accelerator supports a set of generalized vector operations as well as usual scalar operations. Furthermore, specific mathematical functions necessary to the DNC computation are supported in the proposed instruction architecture. As for the microarchitecture, a couple of SRAM modules are employed to fetch operands needed for usual binary ALU operations and its resultant data is written back to one of those SRAM modules. In such operand-fetch and write-back configuration, if the SRAM addressing mode supports a consecutive address generation, those SRAM modules can behave as a sort of vector register file so that this configuration enables us to execute the vector operation by a single instruction, e.g., product-sum, element-wise (Hadamard) multiplication, etc. To extend this mechanism, the proposed microarchitecture also supports a typical neuronal operation executable by a single instruction, i.e., product-sum and non-linear function.

The remaining part of this section describes the proposed instruction set and then explains the microarchitecture with its instruction format and memory allocation scheme. After that, some example programs describing a typical part of the DNC components will be shown.

(a) Instruction set

The instruction set of the proposed accelerator supports two types of data: scalar and vector. Since those data types are freely combined, the instructions can be classified into four categories: an operation from scalars to a scalar, from vectors to a scalar, from vectors to a vector, and from a vector and a scalar to a vector. As for the supported functions necessary to calculate the DNC, arithmetic integer operations including add, subtract and multiplication and lookup-table operations for non-linear functions are provided for both scalar and vector. The supported non-linear functions are sigmoid, hyperbolic tangent, and oneplus. In the case of vector, those operations are executed in element-wise, e.g., Hadamard product. Furthermore, scalar multiplication of scalar and vector, dot product of two vectors, and weighted sum followed by a non-linear function are supported, each of which can be can be carried out by a single instruction. The last one is dedicated to be tuned for several types of neural computation.

The proposed accelerator core executes the target program by combining these functions and data types. There are two types of the instruction formats, i.e., the one is S format for the scalar operations and the other one is V format for vector operations. The V format is shown in Fig. 2. By using these format, all operands and result data can be fetched from or written back to the SRAM modules. In order to calculate memory address of the SRAM flexibly, the addressing mode is register indirect.

The time field in the V format indicates the data set at the time step. In the DNC, all significant data within three time steps are used for the forward activating and the backward learning of the controller network. Thus, the time field denotes a global base address for those data set. The base_x and base_w indicate an operand register which holds the base address of the corresponding variable in the DM_X and DM_W respectively. Furthermore, this register also holds the number of elements in the vector variable n_v. The base_wb indicates a write-back register which holds the base address in the DM_X or DM_W. The flag in the sel field shows the write-back to DM, i.e., DM_X or DM_W. The #elem field indicates an off-set of the memory address. As a result, the memory address can be calculated as follows.

$$Mem.addr. = \{time, base_x\} + \#elem \times n_v \tag{11}$$

where {time, base_x} denotes an operation of bit concatenation. If a vector with n_v elements is stored in the SRAM, the #elem field should be zero. If a 2D matrix are stored, the #elem field should be set from zero to the column width of the matrix. Thus, a matrix-vector product can be described by a set of instructions. In this case, the number of necessary instructions is equal to the column size.

Table 1 shows some typical instructions introduced in the proposed accelerator. For example, TAHv instruction indicates to lookup the table for pre-calculated hyperbolic tangent for each element of the specified vector. NES applies the sigmoid function to the result of product-sum operation of two operand vectors. Similarly, NET applies the hyperbolic tangent. PSM instruction is used to execute the product-sum operation of two operand vectors. MULv denotes the element-wise multiplication of two vectors. If it is repeatedly used for a scalar and a vector, the outer product of vectors can be carried out.

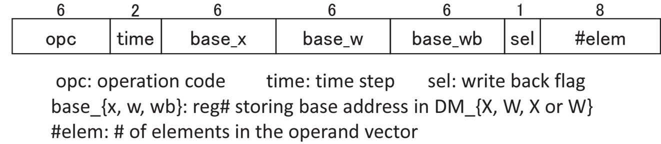

Fig. 2: Instruction format.

Table 1: Example Instructions.

Mnemonic	Description
ADDv	Element-wise addition
SUBv	Element-wise subtraction
MULv	Element-wise multiplication
TAHv	Element-wise table lookup
NES	Neuron operation with sigmoid function
NET	Neuron operation with tanh
PSM	Product-sum

(b) Microarchitecture

The proposed circuit fetches the operands from the two SRAM modules in parallel, executes specified operation indicated by its operation code, and then writes back its result to either SRAM module as shown in Fig 3. Since the operand value is read simultaneously from DM_X and DM_W when executing the instruction, the corresponding operand is stored in different SRAM. For example, to multiply a set of the neuron's pre-synaptic signals by a set of the corresponding weights, the former operand vector is stored in DM_X and the latter operand vector is stored in DM_W. The first memory address is calculated along with the equation 11 and the remaining addresses of the vector is incremented at the second pipeline stages consecutively. During this address increment, the program counter (PC) is not incremented. When the final element address of the vector is generated here, the PC is incremented so as to fetch the next instruction. At the same time, the final result of the product-sum is transferred to the final pipeline stage and lookups the sigmoid function table. Finally, the result of the lookup operation is written back to DM_X as its post-synaptic signal.

The memory allocation for executing the LSTM program over the Eq. 1∼5 is shown in Fig. 4. α denotes the size of the input vector of layer l containing χ_t, i.e., $X + RW + N_l + N_{l-1} + 1$.

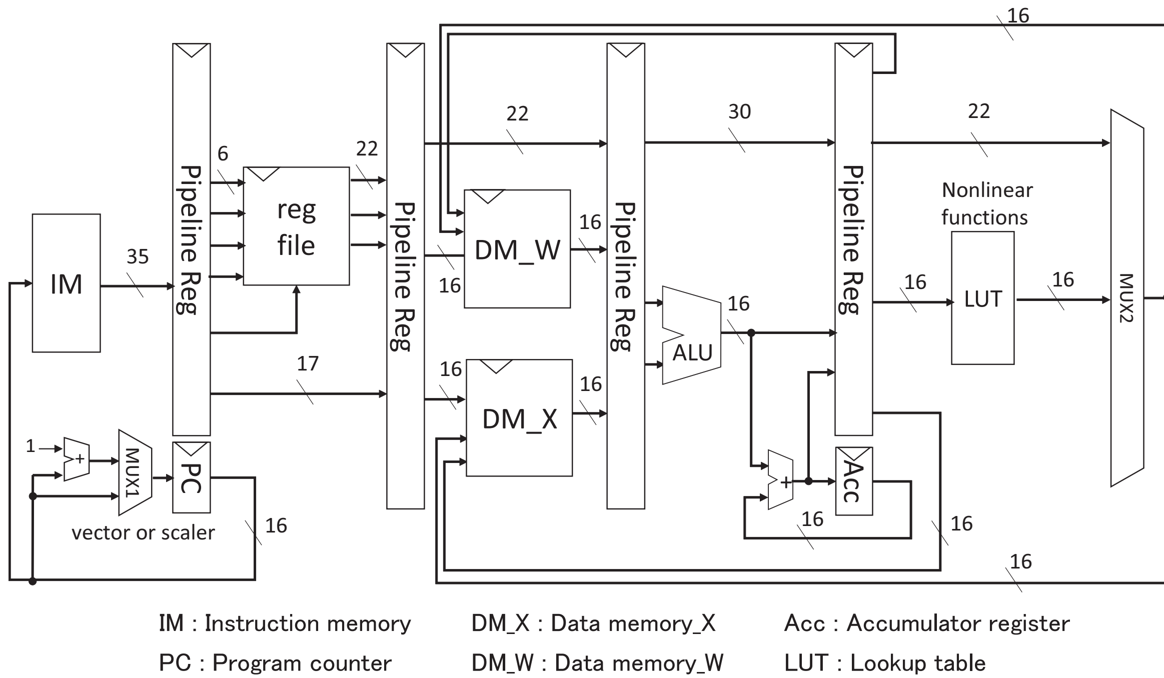

Fig. 3: Structure of accelerator for DNC.

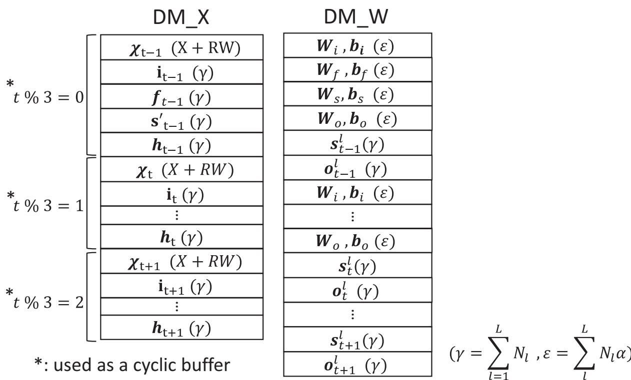

Fig. 4: Memory allocation for the LSTM.

It holds data for three times for LSTM learning and overwrites new data with the oldest time data each time the time advance. Since values are read from two memories at once, it is necessary to store data to be calculated in different memories. s_t' is an area for storeing intermediate calculation results for deriving s_t.

(c) Typical programs

Fig 5 shows an example program of the forward activation process for the l-th layer LSTM network. Here, the number of neurons in this layer is N_l. The first instruction calculates the input gate activation signal of the i-th neuron. After this instruction is repeatedly executed in N_l times, the input gate activation vector $\boldsymbol{i}_t^l$ at the time t can be acquired. Similarly, other gate activation vectors can be calculated. As for the state vector $\boldsymbol{i}_t^l$, in addition to the neuron instruction, an element-wise multiplication instruction and an element-wise addition instruction are needed. Moreover, as for the output activation vector, an element-wise lookup instruction and an element-wise multiplication instruction are required.

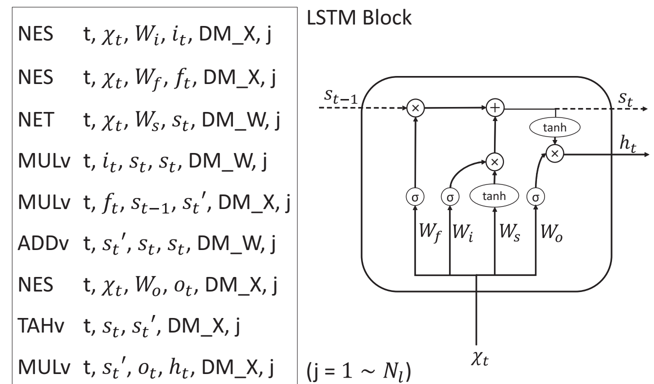

Fig. 5: Example program of the l-th layer LSTM.

Fig 6 shows other program examples. One is for the read operation and the other is for the write operation to the external memory of the DNC. As defined in the equation

9 and 10, these program needs to deal with a transposed matrix. However, these read and write operations can be described by only the continuous addressing because the external memory matrix $\mathrm{M} \in \mathbb{R}^{N \times W}$ is always stored in the transposed form. Even if the transposed matrix $M_t^\top$, the part of the write operation $M_{t-1} \circ (E - \boldsymbol{w}_t^w \boldsymbol{e}_t^\top)$ can be described in the continuous addressing because its operator is the element-wise multiplication.

In this way, the DNC computation can be described by the vector instruction set of the proposed accelerator and it will be efficiently executed in the instruction pipeline mechanism.

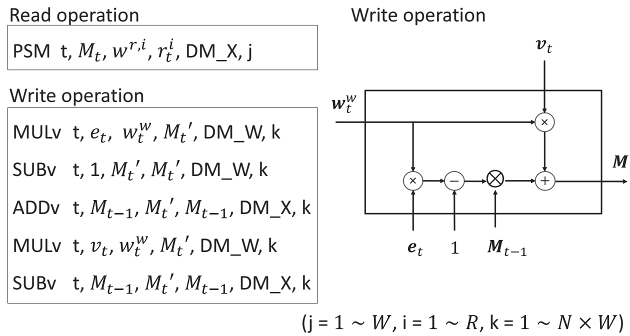

Fig. 6: Example programs of read and write operations.

4. Performance Estimation

In the DNC, the content to be written to the external memory is determined based on the value obtained in the controller network, and the written contents are referred again at next time step. Therefore, in each time step, the forward activating, backward learning, writing to the external memory, and reading from the external memory are computed. In the following estimation, the execution time of each computation is defined $T_{forward}$, $T_{backward}$, T_{write}, and T_{read} respectively.

As mentioned in the previous section, the proposed accelerator core is configured in 5-stage instruction pipeline structure. Therefore, the execution time of the scalar instruction T_s is 5 clock cycles and its IPC is 5. As for the vector instruction, the execution time is (n+4) clock cycles in the case of n-element vectors. This can be expressed $T_v(n)$. The IPC of the vector instruction is inversely proportional to the size of vectors. Since the proposed circuit does not support instructions that perform matrix operations at once, a matrix operation is performed by repetition of vector instructions. Therefore, the execution time of a multiplication of $m \times n$ matrix and m-element vector is estimated at $m\dot{T}_v(n)$. Based on those assumptions, the total execution time of the DNC on the proposed accelerator can be estimated as follows.

(a) Controller network (forward)

The forward activating computation of the controller network is based on the deep LSTM architecture. The computation of neurons in the $l-th$ layer can be estimated based on the equations 1 to 5. The execution time T_{cell} required to calculate the value of the memory cell holding the self-state of the LSTM is expressed as follows from the equation 3.

$$T_{cell} = N_l T_v(X + RW + N_l + N_{l-1} + 1) + 3T_v(N_l) \quad (12)$$

The first term is derived from so-cold weighted sum operations defined in the equation 3, where the input vector in the $l-th$ layer is composed of χ_t, h_{t-1}^l, and h_t^{l-1}. The size of each vector is $X + RW$, N_l, and N_{l-1} respectively. 1 denotes the bias. number of elements of b_s^l. The second term of this equation is derived from three times element-wise operations of N_l-element vectors (neurons).

As for the other equations, 1, 2, 4, and 5, each computation time can be represented by the the first term of the above equation. Therefore, the execution time of the LSTM T_{LSTM} can be calculated as follows including T_{cell}.

$$T_{LSTM} = \sum_{l=1}^{L} \{4N_l T_v(\alpha) + 5T_v(N_l)\} \quad (13)$$

where α denotes $(X + RW + N_l + N_{l-1} + 1)$.

Equations 6 and 7 are used to determine the temporal output vector ν_t and the interface vector ξ_t output from the controller network. The time required for those calculations depends on the total number of neurons constituting the controller network. Equation 9 is used to calculate the output vector of the DNC. The time required for these processes and the LSTM process are integrated into $T_{forward}$.

$$\begin{aligned} T_{forward} =& T_{LSTM} + 2T_v(Y) + YT_v(RW) \\ &+ (Y + WR + 3W + 5R + 3)T_v(\sum_{l=1}^{L} N_l) \end{aligned} \quad (14)$$

(b) Controller network (backward)

In this paper, we assume to use the back-propagation through time (BPTT) for the backward learning of the LSTM. The time $T_{cellback}$ required for the error propagation of a memory cell is expressed as follows.

$$T_{cellback} = 3T_v(\alpha) \quad (15)$$

This equation expresses the execution time for the element-wise multiplications of δy_t, $\delta h_t(s^t)$ and $\delta\sigma'(\hat{o}^t)$ after applying the derivative of the sigmoid function to the vector o_t having α elements. Additionally, the computation time for error propagation to each gate and the updated value

of the weight is needed in the backward learning. $T_{backward}$ is thus expressed by the following equation.

$$\begin{aligned} T_{backward} = \sum_{l=1}^{L} & N_l(3(16T_v(\alpha) + T_v(Y)) \\ & + T_v(\alpha + Y) + T_{LSTM}) \\ & + (14T_v(\alpha) + T_v(Y) + T_v(\alpha + Y) \\ & + T_{LSTM})) \end{aligned} \tag{16}$$

In the backward learning of the LSTM, there are several learnable weight matrices: input gate, output gate, forget gate, normal input, recurrent weight, and bias. The above equation includes the computation time for calculating a matrix of weight errors for one layer corresponding to each gate, and for accumulating this matrix from the second layer to the L-th layer.

(c) Writing to external memory

Writing to the external memory is represented in the equations 10. The computation time of this write weighting $T_{Wweight}$ is expressed by the following equation.

$$T_{Wweight} = 4T_v(N) + T_s \tag{17}$$

where T_C denotes the weighting time for the content-based addressing and T_D denotes the calculation time for the cosine similarity between the write key vector and the current state of the external memory. They are expressed as follows.

$$T_C = NT_D + 5NT_s \tag{18}$$

$$T_D = 3T_v(W) + 5T_s \tag{19}$$

The total computation time for the write operation including the weighting $T_{Wweight}$ is derived as follows.

$$T_{write} = T_C + T_s + 2(R + 4)T_v(N) \tag{20}$$

Where we assume that the execution time T_s for the scalar instructions can be applied for the reciprocal operation, the square root operation, and exponential operation. This is because the domain of operand data is limited so that some of simple approximation algorithms or small lookup table can be introduced into our hardware accelerator.

(d) Reading from external memory

Reading from external memory is represented by the equations 9. In the DNC, there are three types of reading methods: backward weighting, forward weighting, and content weighting. In the above estimation, only the content weighting is assumed to be used. In this case, the computation time for the read weighting and operation T_{read} is expressed as follows.

$$T_{read} = RNT_v(W) + RT_v(N) + RT_C \tag{21}$$

The first term represents the time required to generate a read vector with W elements by using R read heads. The second is for calculating the content weight. The third term ifs for calculating the read weight matrix based on the content weighting.

(e) FPGA Implementation

In order to evaluate the proposed accelerator core, we designed its FPGA circuit and measure the circuit cost and the maximum frequency. The designed circuit supports a partial set of significant instructions described in the previous section. The width of all data is set to 16 bits because it is enough to convergent usual deep neural network as reported in [1]. The capacity of DM_X and DM_W are 1K words and 4K words so that this circuit simulates a maximum of about about 24 × 24 external memory with up to about 70 neurons and 2 read heads.

The designed circuit was compiled by Quartus, Intel Corp. by targeting an Intel FPGA device, Stratix V. The utilization of FPGA circuit resource and the maximum number of executable instructions per second are summarized in Table 2.

Table 2: Logic synthesis result.

FPGA family	Stratix V
Device	5SGSMD4E2H29I2
logic resource	424 / 135,840 (<1%)
Memory [bits]	101,120 / 19,599,360 (<1%)
Max. clock freq.[MHz]	143MHz

Based on the table, 143 M scalar instructions per second can be executed and 8.9 M vector instructions per second can be executed in the case of 16-element vectors. Based on those actual performance, the total execution time of the DNC on the proposed accelerator can be estimated by the equation 14, 16, 20, 21. Since the circuit designed in this research is single core circuit, the processing time can be further shortened by multicore. In that case, considering the FPGA resource consumption of the proposed circuit, there is a possibility that it can be realized as a many core circuit of several tens to several hundreds of cores scale.

5. Conclusion

In this research, we study a hardware accelerator core circuit for the DNC. The proposed accelerator is structured by a kind of vector instruction pipeline. Furthermore, it introduces a set of vector instructions specific to typical neural processing and the DNC. In the paper, the execution time of the DNC is estimated in detail and the FPGA circuit design result of the simplified accelerator is reported.

In order to achieve higher performance for the DNC, we have to investigate data parallel architecture and many core architecture as future works. Furthermore, the calculation precision of the hardware accelerator is also very important in deep neural network applications so that it is a significant work as well as achieving higher performance.

Acknowledgement

Although it is impossible to give credit individually to all those who organized and supported our project, the authors would like to express their sincere appreciation to all the colleagues in the project.

References

[1] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernandez-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 267–278, 2016.

[2] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An ultra-low power convolutional neural network accelerator based on binary weights," in *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*. IEEE, 2016, pp. 236–241.

[3] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 14–26. [Online]. Available: https://doi.org/10.1109/ISCA.2016.12

[4] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwinska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. P. Badia, K. M. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom, K. Kavukcuoglu, and D. Hassabis, "Hybrid computing using a neural network with dynamic external memory," *NATURE*, vol. 538, pp. 471–476, 2016.