Article
AERO: A 1.28 MOP/s/LUT Reconfigurable Inference Processor for Recurrent Neural Networks in a Resource-Limited FPGA

Jinwon Kim, Jiho Kim and Tae-Hwan Kim

School of Electronics and Information Engineering, Korea Aerospace University, Goyang-si 10540, Korea; kjinw1413@kau.kr (J.K.); jhkim_ms@kau.kr (J.K.)
* Correspondence: taehwan.kim@kau.ac.kr

Abstract: This study presents a resource-efficient reconfigurable inference processor for recurrent neural networks (RNN), named AERO. AERO is programmable to perform inference on RNN models of various types. This was designed based on the instruction-set architecture specializing in processing primitive vector operations that compose the dataflows of RNN models. A versatile vector-processing unit (VPU) was incorporated to perform every vector operation and achieve a high resource efficiency. Aiming at a low resource usage, the multiplication in VPU is carried out on the basis of an approximation scheme. In addition, the activation functions are realized with the reduced tables. We developed a prototype inference system based on AERO using a resource-limited field-programmable gate array, under which the functionality of AERO was verified extensively for inference tasks based on several RNN models of different types. The resource efficiency of AERO was found to be as high as 1.28 MOP/s/LUT, which is 1.3-times higher than the previous state-of-the-art result.

Keywords: accelerator architectures; field programmable gate arrays; microarchitecture; neural network hardware; recurrent neural networks

1. Introduction

Recurrent neural networks (RNN) are a class of artificial neural networks whose dataflows have feedback connections. Such recurrent dataflows enable inference to be performed in a stateful manner that is based on not only the current but also past inputs, thereby, recognizing the temporal characteristics [1]. Due to this feature, the RNN inference can be employed in diverse applications that require the handling of sequential or time-series data, such as in language modeling [2], sequence classification [3], and handwriting recognition [4]. However, the computational workload involved in the RNN inference is often intractably high for practical models. Hence, a dedicated hardware to accelerate the inference process is necessary, and its efficiency is crucial when implemented using resource-limited field-programmable gate arrays (FPGAs).

There are various types of RNN models, and their characteristics in the workload and achievable inference performance levels are different. There is no unique model type that can achieve the best inference performance with the lowest workload for every application. Therefore, a model type needs to be selected considering the intended applications and their design objectives. Envisioning the processing of the RNN inference of different models for various applications with different design objectives, we need an efficient RNN inference processor to support the model reconfigurability.

There are several previous studies regarding the design and implementation of efficient RNN inference processors using FPGAs. Most of the previous RNN inference processors were designed to support only one type of model: some of them can perform RNN inference based only on long short-term memory (LSTM) [5], as LSTM is generally beneficial to achieve good inference performance in particular for tasks relying on long-term dependencies [6–11]; others employed the gated-recurrent unit (GRU) [12] to achieve...
more efficient architectures [13,14]; and an efficient processor to accelerate the training of the vanilla-RNN-based language model was presented in [15]. An FFT-based compression technique for the RNN models and a systematic design framework based on this technique were proposed in [10,16].

A GRU inference system was developed by integrating dedicated matrix computing units [13]. An efficient architecture to perform the GRU inference was presented based on the modified model exploiting the temporal sparsity [17]. A reconfigurable system presented in [18] was designed to perform inference based on LSTM as well as convolutional neural networks. As the multiplications are compute-intensive kernels involved in the RNN inference, a previous work attempted to approximate them based on a technique motivated by stochastic computing [7].

This study presents an efficient RNN inference processor named AERO. AERO is an instruction-set processor that can be programmed to perform RNN inference based on models of various types, where its instruction-set architecture (ISA) is formulated to efficiently perform the common primitive vector operations composing the dataflows of the models. AERO is designed by incorporating a versatile vector-processing unit (VPU) and utilizing it to perform every vector operation consistently while achieving a high resource efficiency.

To reduce the resource usage, multiplications are carried out approximately without affecting the inference results noticeably, and the number of the tables in the activation coefficient unit (ACU) is reduced by exploiting the mathematical relation between the activation functions. We verified the functionality of AERO for inference tasks based on several different RNN models under a fully integrated prototype inference system developed using Intel® Cyclone®-V FPGA. The resource usage to implement AERO was 18,000 LUTs, and the inference speed was 23 GOP/s, which shows the resource efficiency of 1.28 MOP/s/LUT.

The rest of the paper is organized as follows. In Section 2, we analyze the dataflows of the RNN models of various types. In Section 3, we describe the ISA and microarchitecture of AERO in detail. Section 4 presents the implementation results and provides the evaluation in comparison to the previous results. In Section 5, we draw our conclusions.

2. Dataflow of RNN Inference

RNN models have recurrent dataflows formed by feedback connections such that inference can be performed effectively based on the states affected by the past input. Figure 1 illustrates the dataflow of the traditional vanilla RNN model [19] along with those of the advanced variants [5,12]. The elementwise multiplication of the vectors \( \mathbf{a} \) and \( \mathbf{b} \) is represented by \( \mathbf{a} \times \mathbf{b} \). Each model contains one or more fully-connected layers followed by non-linear activation functions, which regulate the propagation of the information from the current input and state to the next state. Although the dataflows of the models are dissimilar to each other, they can be described by a few common primitive vector operations such as matrix-vector multiply-accumulate (MAC), elementwise MAC, and activation functions.

The RNN models are different from each other with respect to the computational workload and achievable inference performance. Table 1 illustrates the workload and inference performance of the three RNN models of different types designed targeting the sequential MNIST tasks [20] through different steps. In the sequential MNIST tasks, an image is segmented by the number of steps, and each segment is input to the models for each step as described in [20]. The images in the original dataset were resized to 32 × 32 for the purpose of convenient segmentation. In estimating the workload, the addition and multiplication was counted by one OP and two OPs, respectively.

The trade-off between the workload and inference performance can be found in Table 1. Since there is no unique model type that always outperforms the others in terms of both workload and performance in an unparalleled way, the model design, including the selection of its type, needs to be carefully made subject to the application-specific
objectives and constraints. For example, LSTM is more favorable to achieve a superior inference performance compared with the vanilla RNN or GRU. However, the vanilla RNN or GRU might be efficient owing to the low workload when applied to certain tasks that do not rely on long-term dependencies (e.g., the sequential MNIST task through 16 steps in Table 1). This is the motivation for AERO to support the reconfigurability for the models of various types.

![Dataflow graphs of the recurrent neural network (RNN) models](image)

**Figure 1.** Dataflow graphs of the recurrent neural network (RNN) models, where x, h, and c represent the input activation, hidden state, and cell state vectors, respectively. W and b represent the weight matrix and bias, respectively. The subscripts are used to distinguish the gates.

**Table 1.** Workload and achievable accuracy of the RNN models for sequential MNIST tasks, where the state size of the models is 128.

| Number of Steps | RNN Model Type | Workload (KOP/Step) | Accuracy (%) |
|-----------------|----------------|---------------------|--------------|
|                 | Vanilla RNN    | 73                  | 98.11        |
|                 | GRU            | 222                 | 98.83        |
|                 | LSTM           | 296                 | 98.86        |
| 32              | Vanilla RNN    | 70                  | 97.14        |
|                 | GRU            | 218                 | 98.80        |
|                 | LSTM           | 292                 | 98.84        |
| 64              | Vanilla RNN    | 66                  | 73.98        |
|                 | GRU            | 210                 | 98.19        |
|                 | LSTM           | 288                 | 98.47        |
3. Proposed Processor: AERO
3.1. RNN-Specific Instruction-Set Architecture

The ISA of AERO is formulated with the objective of efficiently performing the primitive vector operations that compose the dataflows of RNN models. The ISA defines a special data type known as the vector, which is the basic unit of the dataflow processing in AERO. Each vector is composed of $P$ $w$-bit elements and stored in a memory. Several memories store the vectors, namely, the activation memory (AM), weight memory (WM), and bias memory (BM), which are appropriately named to express their purpose and addressable by $w$ bit. The instruction memory (IM) stores the program, which is an instruction list to describe a certain dataflow. The ISA has sixteen pointer registers storing the addresses for the memory accesses, and their roles are summarized in Table 2.

The ISA supports only a few kinds of instructions, some of which can be used for the vector processing while others can be used for the pointer handling. Table 3 describes the behaviors of the supported instructions. The inner product of the two vectors $a$ and $b$ is represented by $a \odot b$. The bitwise shift, or, and inversion operators are represented by $\ll$, $\mid$, and $\neg$, respectively. SignExt($\cdot$) and ZeroExt($\cdot$) extend the signed and unsigned input operands, respectively. MVMA, EMAC, and ENOF belong to the vector-processing instructions and have complex behaviors that realize the primitive vector operations composing the dataflows through several microoperations, as described in Table 3.

Furthermore, they directly use the vector operands stored in the memories according to the register-indirect addressing. The ISA provides a simple programming model such that each vector-processing instruction corresponds directly to each primitive vector operation, reducing the instruction count involved to describe a dataflow. CSL, SHL, ACC, and SAC belong to the pointer-handling instructions. They provide the simple arithmetic and logical operations for efficiently handling the addresses stored in the pointer registers.

3.2. Microarchitecture
3.2.1. Processing Pipeline

AERO was designed based on the proposed RNN-specific ISA with $P = 64$ and $w = 16$. Figure 2 shows the processing pipeline, which is composed of seven stages. In Stage 1, an instruction is fetched from IM. In Stage 2, the control signals are generated by decoding the fetched instruction; the pointers are read for the subsequent memory accesses and possibly updated. In Stage 3, the vector operands are read from one or more memories by the addresses provided by the pointers; ACU finds the coefficients for evaluating the activation functions. In Stages 4–6, VPU processes the vector operands served from the preceding stage. In Stage 7, the resulting vector from VPU is written to the memory (AM). The processing throughput of AERO is basically one vector per cycle. If multiple vector operations are involved in a single vector-processing instruction, it may take multiple cycles to execute the instruction. For example, it takes $(\text{DST\_BOUND} - \text{DST})(\text{SRC0\_BOUND} - \text{SRC0})/64$ cycles to execute a single MVMA instruction.

AERO incorporates a versatile VPU to perform every kind of vector operation. As the dataflow analysis in Section 2 implies, the primitive vector operations that are necessarily supported by AERO are the matrix-vector multiplication, elementwise MAC, and activation functions. VPU either performs the elementwise MAC or computes the inner product of the vectors. The matrix–vector multiplication is performed by the VPU computing the inner products iteratively with the vectors. The activation functions are evaluated by employing a linear spline, for which the elementwise MAC is also performed by VPU.

By utilizing the VPU in this manner to efficiently perform every kind of vector operation, AERO can achieve a high resource efficiency. In contrast, many of the previous RNN inference processors, including those presented in [6–8], were designed based on an architecture that incorporates multiple different processing units, each of which can perform a certain vector operation only. This might be inefficient in terms of the resource efficiency because some of the processing units may not perform any operations inevitably due to the data dependency inherently imposed by the dataflows.
Table 2. Pointer registers in AERO.

| Register | Alias  | Role                           |
|----------|--------|-------------------------------|
| R0       | DST    | Destination address in AM    |
| R1       | SRC0   | First source address in AM   |
| R2       | SRC1   | Second source address in AM  |
| R3–R7    | -      | Placeholders                 |
| R8       | BIAS   | Bias address in BM           |
| R9       | WEIGHT | Weight address in WM         |
| R10      | DST_BOUND | Bound of the destination address |
| R11      | SRCO_BOUND | Bound of the first source address |
| R12–R15 | -      | Placeholders                 |

Table 3. Instructions in AERO.

| Instruction | Function | Behavior | Format |
|-------------|----------|----------|--------|
| MVMA        | Matrix-vector MAC | base ← SRCO while DST < DST_BOUND do bias ← BM[BIAS] while SRCO < SRCO_BOUND do in0 ← AM[SRCO] in1 ← WM[WEIGHT] AM[DST] ← in0 × in1 + bias SRCO ← SRCO + P WEIGHT ← WEIGHT + P end while BIAS ← BIAS + 1 DST ← DST + 1 SRCO ← base end while | 14 6 5 4 3 0 | Reserved 0 000 |
| EMAC.acc.inv | Elementwise MAC, where acc indicates that the result is accumulated and inv indicates the bitwise inversion of the first operand. | while DST < DST_BOUND do in0 ← in0 + AM[SRC0] in1 ← AM[SRC1] in2 ← acc ? AM[DST] : 0 AM[DST] ← in0 × in1 + in2 DST ← DST + P SRC0 ← SRC0 + P SRC1 ← SRC1 + P end while | 14 6 5 4 3 0 | Reserved 0 011 |
| ENOF.type   | Elementwise non-linear function, where type indicates the function type. | while DST < DST_BOUND do AM[DST] ← function values of AM[SRC0] DST ← DST + P SRC0 ← SRC0 + P end while | 14 6 5 4 3 0 | Reserved 0 010 |
| CSL ra, imm8 | Constant load, where \(a \in \{0, 1, \ldots, 15\}\) ra ← ZeroExt(imm8) and imm8 is given by the 8-bit immediate constant. | 14 11 10 3 2 0 | ra imm8 100 |
| SHL ra, imm8 | Shift and load, where \(a \in \{0, 1, \ldots, 15\}\) ra ← (Ra ≪ 8) | ZeroExt(imm8) and imm8 is given by the 8-bit immediate constant. | 14 11 10 3 2 0 | ra imm8 101 |
| ACC ra, rb, imm4 | Accumulate, where \(a, b \in \{0, 1, \ldots, 15\}\) ra ← rb + SignExt(imm4) and imm4 is given by the 4-bit immediate constant. | 14 11 10 7 6 3 2 0 | ra b imm4 110 |
| SAC ra, rb, imm4 | Shift and accumulate, where \(a, b \in \{0, 1, \ldots, 15\}\) ra ← rb + (SignExt(imm4) ≪ \(\log_2{P}\)) and imm4 is given by the 4-bit immediate constant. | 14 11 10 7 6 3 2 0 | ra b imm4 111 |
3.2.2. Vector Processing Unit Based on the Approximate Multipliers

VPU is designed to achieve a low resource usage. Figure 3 shows the microarchitecture of VPU, in which the two highlighted datapaths are the ones through which the vector operations (elementwise MAC and inner product computation) are performed. The microarchitecture is designed to allow the two paths to share several components for the purpose of reducing the resource usage; more specifically, the multipliers and adders in the first two stages of the VPU are shared by the two paths, which are drawn by the solid and dotted lines in Figure 3. The summation unit in the third stage computes the sum of the 33 inputs based on the Wallace tree, whereby the accumulation involved in computing the inner product is carried out.

Each multiplier in the first stage of VPU carries out the multiplication of the 16-bit two’s complement operands on the basis of an approximation scheme. A 16-bit two’s complement operand, which is denoted by $x$, can be truncated to $x[7:0]$ without any loss if $x[15:7]$ has the pattern of all zeros or ones. Here, $x[i:j]$ stands for the sub bit-vector of $x$ ranging from the $i$-th to the $j$-th bit.

Exploiting such truncatability, the proposed scheme carries out the 16-bit $\times$ 8-bit exact multiplication to obtain the approximate result of the 16-bit $\times$ 16-bit multiplication, as described in Table 4, and the multiplier design based on the proposed scheme is shown in Figure 3. The prefix 0x of the number literals stands for the hexadecimal representation. The proposed scheme reduces the resource usage considerably because it entails only half the number of the partial products compared to that for the exact multiplication, considering that the number of the partial products of $a$-bit $\times b$-bit is in $O(ab)$.

The proposed approximation scheme does not affect the inference results noticeably. The cases that make an operand truncatable in the proposed scheme corresponds that the operands have the values near zero since the operand is represented by the two’s complement format. These cases are probable in practice. Figure 4 illustrates the practical operand distributions aggregated while performing the RNN inference for the sequential MNIST task, in which we can find that most of the operands have values near zero. The probability of the first two cases in Table 4, for which no approximation error will be brought about by producing the exact multiplication results, is at least 0.49 in every model used to obtain the results in Figure 4.

This is much higher than the probability calculated assuming the uniform distribution, $1 - (1 - 2 \cdot (1/2)^9) \cdot (1 - 2 \cdot (1/2)^9) \approx 0.008$. In other cases, the multiplication is performed...
in such a way to not account for the partial products related to the insignificant bits of the operands, as described in Table 4, and the inference results are, thus, not affected significantly. In the sequential MNIST task to obtain the results in Figure 4, the accuracy loss caused by the approximation is below 0.7%.

**Figure 3.** Microarchitecture of the vector processing unit.

**Figure 4.** Distributions of the multiplier operands in the RNN inference for the sequential MNIST task through 16 steps based on (a) GRU, (b) LSTM, (c) peephole LSTM [21], and (d) bidirectional LSTM models [22], whose state sizes are 64, 96, 64, and 64, respectively.
Neither $x$ is truncatable and $y$ is truncatable.

Neither $x$ nor $y$ is truncatable and $x[7]$ is on.

Neither $x$ nor $y$ is truncatable and $x[7]$ is off.

Additional remarks that are worth noting:

- The truncation is performed by dropping the upper eight bits of an operand in the proposed multiplication approximation scheme. The truncation is performed in a consistent manner without regard to the RNN models and, thus, can be fulfilled by a simple logic circuitry picking the sub bit-vector at the fixed position as shown in Figure 3.

- A different truncation size might be considered in applying the proposed multiplication approximation scheme. When the truncation size is $\tau$, 16-bit $\times$ 16-bit multiplication is carried out by the 16-bit $\times (16 - \tau)$-bit multiplier by dropping out the upper $\tau$ bits in one of the multiplication operands. With a larger $\tau$, the multiplier becomes simpler so that its resource usage can become less. However, this may affect the inference results more severely because the probability that both of the two operands are not truncatable, which correspond to the last two cases in Table 4, brings about the approximation errors, may become larger. $\tau$ was determined to be 8 so that the proposed multiplication approximation scheme does not have a noticeable effect on the inference results, which were validated extensively based on the experimental results.

- The proposed scheme exploits the truncatability of the multiplication operands, which is highly probable in the inference based on the RNN models (e.g., vanilla RNN, GRU, and LSTM) that are already trained. Therefore, it does not entail any training issues necessarily addressed by a special methodology, such as the retraining [6]. It does not require any model modifications, either.

### Table 4. Multiplication approximation scheme.

| Case                          | Product          | Example            |
|-------------------------------|------------------|--------------------|
| $x$ is truncatable.           | $x[7:0] \times y[15:0]$ | $x = 0xFF80, y = 0xABCD$ |
|                               |                  | $\rightarrow xy = 0x80 \times 0xABCD$ |
| $x$ is not truncatable and $y$ is truncatable. | $x[15:0] \times y[7:0]$ | $x = 0x1234, y = 0x007D$ |
|                               |                  | $\rightarrow xy = 0x1234 \times 0x7D$ |
| Neither $x$ nor $y$ is truncatable and $x[7]$ is on. | $x[15:0] \times y[15:8] \ll 8$ | $x = 0x12F4, y = 0x0BCD$ |
|                               |                  | $\rightarrow xy \approx 0x12F4 \times 0x0B \ll 8$ |
| Neither $x$ nor $y$ is truncatable and $x[7]$ is off. | $x[15:8] \times y[15:0] \ll 8$ | $x = 0xAB12, y = 0xABCD$ |
|                               |                  | $\rightarrow xy \approx 0xAB \times 0xABCD \ll 8$ |

* A 16-bit two’s complement number $p[15:0]$ is truncatable to $p[7:0]$ if $p[15:7]$ has the pattern of all zeros or ones.

3.2.3. Activation Coefficient Unit Based on the Reduced Tables

The non-linear activation functions are evaluated by employing a linear spline. The sigmoid function of $x$, which is denoted by $\sigma_g(x) \equiv 1/(1 + e^{-x})$, is evaluated by

$$a(x) \cdot (x - \kappa(x)) + \beta(x),$$

where $\kappa(x)$ represents the knot, which is the left end of the segment belonging to $x$, and $a(x)$, and $\beta(x)$ represent the coefficients corresponding to the slope and offset of the segment, respectively. $x$ is represented by a 16-bit two’s complement number, and $\kappa(x)$ is determined as $x[15:12]$, so that $x - \kappa(x)$ is simplified to $x[11:0]$. ACU finds $a(x)$ and $\beta(x)$ by looking up the tables storing the pre-computed slopes and offsets with the index given by $\kappa(x)$ for the subsequent MAC operation to be performed by VPU.

Another activation function, the hyperbolic tangent function, has to be supported additionally in order to process the dataflows of the models of various types. Furthermore, such a coefficient lookup is executed for every element composing a vector in parallel; for this purpose, we need as many tables as the number of the elements in a vector. Therefore, the resource usage involved to implement ACU is not negligibly small.

ACU is designed to have no additional tables storing the coefficients for the hyperbolic tangent function; it finds the coefficients for the hyperbolic function by modifying those
for the sigmoid function based on the mathematical relation between the functions. Let us denote the hyperbolic tangent function of $x$ by $\sigma_t(x) \triangleq (e^{2x} - 1)/(e^{2x} + 1)$. Since $\sigma_t(x)$ is equal to $2\sigma_e(2x) - 1$, it can be evaluated using (1) by

$$2\alpha(2x) \cdot (2x - \kappa(2x)) + 2\beta(2x) - 1. \quad (2)$$

Here, $\alpha(2x)$ and $\beta(2x)$ can be obtained by looking up the tables for the sigmoid function with the index determined considering the saturation as follows:

- $0\text{b}1000$ for $x[15] = 0\text{b}1$,
- $0\text{b}0111$ for $x[15 : 14] = 0\text{b}01$,
- $x[14 : 11]$ for other cases,

where the prefix $0\text{b}$ of the number literals stands for the binary representation. Figure 5 shows the microarchitecture of ACU. It should be remarked that $2\beta(2x) - 1$, which is the offset in evaluating $\sigma_t(x)$, is realized by the simple logical operation as shown in Figure 5 since $0\leq\beta(2x)<1$. When compared with the straightforward architectures, including those presented in [6–10,16,17,23], which were designed without exploiting the mathematical relation between the functions, the number of the tables for the proposed scheme can be reduced by as much as half due to the shared usage of the tables. This leads to a reduction of the logic resource usage for ACU by 29% in terms of the LUT count in the ACU implementation results.

3.3. Prototype Inference System

A prototype RNN inference system was developed to verify the functionality of AERO using an FPGA. Figure 6 describes the overall architecture of the inference system into which all the essential components, including the MCU, are integrated. The memories that are associated directly with AERO, i.e., AM, WM, BM, and IM, were designed by instantiating BRAMs directly based on the structures shown in the figure considering the required bandwidths by AERO. The bandwidths provided by WM and AM required to avoid stalling the pipeline of AERO are $64 \times 16$ bits/cycle and $64 \times 16 \times 4$ bits/cycle, respectively.

To realize such high bandwidths, WM and AM were built based on the multi-bank structures of the BRAM instances; specifically, AM has been designed by incorporating the access router that is capable of routing the data transfers dynamically from/to the internal dual-port BRAM instances organized based on the multi-bank structure. The architecture of the system has been described using HDL to be synthesized targeting an FPGA device.
The inference procedure is actualized using the components in the system according to the illustration in Figure 7. MCU preloads the dataflow description program, which was created based on the ISA of AERO, into IM, and the weight matrices and bias vectors into WM and BM, respectively. MCU and AERO run in a lock-step manner for each step as illustrated in the figure; MCU feeds the input activation vector to AERO by loading it to AM, and AERO runs the inference. They can work in parallel since the part of AM that stores the input activation vector is designed to support the double-buffering scheme. Finally, the inference results are demonstrated via the parallel IO and VGA subsystem.

![Figure 6. Overall architecture of the prototype inference system.](image)

![Figure 7. Overall inference procedure for N steps in the prototype inference system.](image)

### 4. Results and Evaluation

The prototype RNN inference system based on AERO was synthesized using Intel® Quartus® Prime v20.1 targeting Intel® Cyclone®-V FPGA (5CSXF6D6). The entire system was successfully fitted in such a resource-limited FPGA device, utilizing the resource usage of 27,000 LUTs, 2653 Kbit BRAMs, and 68 DSPs. The resource usage of AERO is just 18,000 LUTs, 1620 Kbit BRAMs, and 64 DSPs, where the BRAMs were used to implement AM, WM, BM, and IM. Here, the LUT count was estimated to be the ALUT [24] count in the target device, as suggested by the guidelines in [25]. The maximum operating frequency...
of the system was estimated to be 120 MHz under the slow model with a 1.1 V supply at 85 °C, at which the peak inference speed is as high as 23 GOP/s, and the average power consumption is 138.3 mW.

The functionality of AERO was verified successfully by programming it to perform inference tasks based on the various RNN models listed in Tables 5 and 6 for the sequential MNIST tasks through different steps [20] and the word-level Penn Treebank task [26]. The inference performance (i.e., the inference accuracy in the sequential MNIST task and the perplexity in the Penn Treebank task) was obtained for the fixed-point models associated with the proposed multiplication approximation (in Section 3.2.2) and table reduction schemes (in Section 3.2.3). The verification environment setup is shown in Figure 8. The demonstration video is accessible via https://youtu.be/nmy8K1bRgII on 24 May 2021.

Table 5. Performance of AERO for the various RNN models targeting the sequential MNIST tasks [20].

| RNN Model Type       | Vanilla RNN | GRU   | LSTM | GRU   | LSTM | Bi-Directional LSTM [22] | Peephole LSTM [21] | Bi-Directional LSTM [22] |
|----------------------|-------------|-------|------|-------|------|---------------------------|-------------------|---------------------------|
| Number of steps      | 16          | 16    | 16   | 32    | 32   | 32                        | 64                | 64                        |
| State size           | 128         | 128   | 128  | 96    | 96   | 96                        | 64                | 64                        |
| Workload (KOP/step)  | 73.73       | 222.34| 295.81| 111.46| 148.13| 296.26                   | 172.93            | 444.16                   |
| Processing latency (µs/step) | 3.24 | 9.70 | 12.93 | 4.88 | 6.50 | 13.00                     | 7.60               | 19.47                     |
| Inference accuracy (%) | 97.32       | 97.91 | 98.47| 97.36 | 97.59| 98.00                    | 97.88             | 97.94                     |
| Normalized resource usage (LUT/step/s) | 0.06 | 0.17 | 0.23 | 0.09 | 0.12 | 0.23                     | 0.14               | 0.35                      |
| Normalized energy consumption (µJ/step) | 0.45 | 1.34 | 1.79 | 0.67 | 0.90 | 1.80                     | 1.05               | 2.69                      |

Figure 8. Verification environment setup for the sequential MNIST tasks.

Providing the reconfigurability, AERO exhibited scalability in normalized resource usage as well as normalized energy consumption to achieve a certain inference performance. AERO can be configured to achieve a superior inference performance with more resource usage and higher energy consumption or a moderate inference performance with less resource usage and less energy consumption when the resource usage and energy consumption are assessed in a normalized fashion.

In Tables 5 and 6, the normalized resource usage was estimated by the usage of the logic resource to achieve the processing speed per step in the inference. The normalized
energy consumption was estimated by the energy consumed per each step in the inference. These metrics are directly related with the latency taken to process the workload of the models. AERO can achieve a superior inference performance by being configured to run the inference based on a complex model; or else, can become more efficient in the resource usage and energy consumption by being configured to run the inference based on a simple model.

The implementation results of AERO are compared with the previous results in Table 7. The previous state-of-the-art RNN inference processors implemented using FPGA devices were selected for fair comparisons. Here, the resource efficiency is defined so that the comparisons can be conducted in a model-neutral way as in the previous study [27]. AERO showed a relatively low resource usage against the other previous processors. However, its inference speed was not very low, thus, leading to a high resource efficiency.

Some previous RNN inference processors [6,10,11,17] showed very high inference speeds effectively by exploiting the model sparsity; however, such a high inference speed is not guaranteed as it is theoretically subject to meet a certain degree of the inference performance even with a special retraining process. The resource efficiency of AERO is 1.3-times higher than the previous best result. This is contributed to by its microarchitecture, which utilizes the VPU in an efficient manner to perform every vector operation; furthermore, its major building blocks, VPU and ACU, were designed based on novel schemes to reduce resource usage.

More importantly, AERO supports reconfigurability to perform the inference based on the RNN models of various types, and this was verified extensively under the prototype system developed to perform the practical inference tasks. To the best of our knowledge, AERO is the first RNN inference processor that has been proven to provide reconfigurability supporting various model types. The energy efficiency of AERO was higher than the previous results in the table. This may be due to the low-power characteristic of the cost-effective FPGA device used in this work; however, such FPGA devices usually have a tight limitation of the available resources, to which AERO has been successfully fitted and showed a high inference speed.

Even though AERO was implemented based on a single processing core based on the architecture presented in the previous section, it may achieve a higher inference speed while maintaining the resource efficiency with additional processing cores integrated. The primitive vector operations in the RNN models of various types (i.e., matrix-vector MAC, elementwise MAC, and elementwise activation) can be decomposed into multiple vector operations of a smaller size.

If the decomposed operations are performed in parallel by multiple processing cores that share a dataflow description program, the inference speed can be increased by a factor of the number of the processing cores. Such parallel processing by multiple cores does not entail any aggregation overhead, and thus the resource efficiency can be maintained. Further studies may be followed to achieve a high inference speed by materializing such architecture.

Table 6. The performance of AERO for the various RNN models targeting the word-level Penn Treebank task [26].

| RNN Model Type | LSTM | Bidirectional GRU [22] | GRU |
|----------------|------|-----------------------|-----|
| State size     | 64   | 64                    | 128 |
| Workload (KOP/step) | 98.75 | 148.61                | 222.34 |
| Processing latency (µs/step) | 4.33  | 6.50                  | 9.70 |
| Perplexity per word | 120.86 | 116.9                 | 108.94 |
| Norm. resource usage (LUT/step/s) | 0.08  | 0.12                  | 0.17 |
| Norm. energy consumption (µJ/step) | 0.60  | 0.90                  | 1.34 |

The previous state-of-the-art RNN inference processors implemented using FPGA devices were selected for fair comparisons. Here, the resource efficiency is defined so that the comparisons can be conducted in a model-neutral way as in the previous study [27]. AERO showed a relatively low resource usage against the other previous processors. However, its inference speed was not very low, thus, leading to a high resource efficiency.
Table 7. Implementation results of the RNN inference processors based on field-programmable gate arrays.

| Inference Processor | AERO  | [6]  | [7]  | [8]  | [9]  | [10] a | [11] | [13] |
|---------------------|-------|------|------|------|------|--------|------|------|
| **FPGA device**     |       |      |      |      |      |        |      |      |
| Name                | Cyclone®-V | Kintex®-UltraScale | Zynq®-7000 | Zynq®-7000 | Virtex®-7 | Virtex®-7 | Arria®-10 | Stratix®-V |
| Part number         | 5CSXFC6D6 | XCKU060 | XCUZ030 | XCUZ020 | XCVVX485T | XCVVX690T | GX1150 | N.A. |
| Model reconfigurability | Yes | No (only LSTM) | No (only LSTM) | No (only LSTM) | No (only LSTM) | No (only LSTM) | No (only LSTM) | No (only LSTM) |
| Inference speed (GOP/s) | 23.0 | 282.2 | 8.08 | 0.29 b | 7.26 | 131.1 | 304.1 | 126.70 |
| Precision c         | FxP-16 | FxP-12 | FxP-8 | FxP-16 | FP | FxP-16 | FxP-16 | FP |
| Resource usage      |       |      |      |      |      |        |      |      |
| LUT (K)             | 18.0 | 293.9 | 23.0 | 7.6 | 198.3 | 504.4 | 578.0 d | 592.0 d |
| BRAM (Kbit)         | 1620 e | 34092 | 6480 | 576 | 38592 | 34768 | 48760 | 4000 |
| DSP                 | 64   | 1504 | 0    | 50  | 1176 | 2675 | 1518 | 256 |
| FF (K)              | 10.1 | 453.1 | 28.4 | 12.9 | 182.6 | 199.7 | N.A. | N.A. |
| Resource efficiency (MOP/s/LUT) | 1.28 | 0.96 | 0.35 | 0.04 | 0.04 | 0.26 | 0.52 | 0.21 |
| Power consumption (W) | 0.77 (0.138) f | 41.00 | 1.19 | 1.94 | 19.63 | 22.00 | 19.10 | N.A. |
| Energy efficiency (GOP/J) | 29.08 (166.31) f | 6.88 | 6.79 | 0.15 | 0.37 | 5.96 | 15.92 | N.A. |

a This corresponds to C-LSTM FFT8 in [10]. b The inference speed in terms of GOP/s is not presented in [8] and is estimated in [7] for the comparison. This result was excerpted here for the same purpose. c FxP-n stands for the precision achievable by the n-bit fixed-point numbers, and FP stands for that achievable by the floating-point numbers. d As no direct results of the LUT counts were found in [11,13], the LUT counts were estimated to be the ALUT [24] counts according to the official guidelines in [25]. The ALUT counts can be obtained from the ALM [24] counts considering the number of the ALUTs in each ALM in the target devices. e This result corresponds to the BRAM instances for implementing AM, WM, BM, and IM, which are associated directly with AERO. f The number inside the parentheses corresponds to the result of AERO itself, while the number outside the parentheses corresponds to that of the entire system.
5. Conclusions

In this study, we presented the design and implementation of a resource-efficient reconfigurable RNN inference processor. The motivation of supporting the reconfigurability by the RNN inference processor was founded by investigating the workloads and inference performance of the various model types. The proposed processor, named AERO, is an instruction-set processor whose ISA was designed to process the common primitive vector operations in the dataflows of the RNN models of various types, thus, achieving programmability for them. AERO utilizes a versatile VPU to perform every vector operation efficiently.

To reduce resource usage, the multipliers in VPU were designed to perform the approximate computations, and the number of the tables in ACU was reduced by exploiting the mathematical relation between the activation functions. The functionality of AERO was successfully verified for the inference tasks based on several different RNN models under a prototype system developed using a resource-limited FPGA. The resource efficiency of AERO was as high as 1.28 MOP/s/LUT. In further studies, efficient RNN inference systems may be developed based on AERO; one example may aim at achieving a higher inference speed by integrating multiple cores and maintaining the resource efficiency of AERO.

Author Contributions: Conceptualization, J.K. (Jinwon Kim) and T.-H.K.; Data curation, J.K. (Jinwon Kim) and J.K. (Jiho Kim); Formal analysis, J.K. (Jinwon Kim) and T.-H.K.; Funding acquisition, T.-H.K.; Investigation, J.K. (Jinwon Kim) and J.K. (Jiho Kim); Methodology, T.-H.K.; Project administration, T.-H.K.; Software, J.K. (Jinwon Kim) and J.K. (Jiho Kim); Supervision, T.-H.K.; Validation, J.K. (Jinwon Kim) and J.K. (Jiho Kim); Visualization, J.K. (Jinwon Kim) and J.K. (Jiho Kim); Writing—original draft, T.-H.K.; Writing—review & editing, J.K. (Jinwon Kim) and J.K. (Jiho Kim) and T.-H.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) [2017-0-00528, The Basic Research Lab for Intelligent Semiconductor Working for the Multi-Band Smart Radar] and the GRRC program of Gyeonggi province [2017-B02, Study on 3D Point Cloud Processing and Application Technology]. The EDA tools were supported by IDEC, Korea.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Abiodun, O.I.; Jantan, A.; Omolara, A.E.; Dada, K.V.; Mohamed, N.A.; Arshad, H. State-of-the-art in artificial neural network applications: A survey. *Heliyon* 2018, 4, e00938. [CrossRef] [PubMed]
2. Athiwaratkun, B.; Stokes, J.W. Malware classification with LSTM and GRU language models and a character-level CNN. In Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2017), New Orleans, LA, USA, 5–9 March 2017; pp. 2482–2486.
3. Jurgovsky, J.; Granitzer, M.; Ziegler, K.; Calabretto, S.; Portier, P.E.; He-Guelton, L.; Caelen, O. Sequence classification for credit-card fraud detection. *Expert Syst. Appl.* 2018, 100, 234–245. [CrossRef]
4. Graves, A.; Liwicki, M.; Fernández, S.; Bertolami, R.; Bunke, H.; Schmidhuber, J. A novel connectionist system for unconstrained handwriting recognition. *IEEE Trans. Pattern Anal. Mach. Intell.* 2008, 31, 855–868. [CrossRef] [PubMed]
5. Stepp, H.; Jurgen, S. Long short-term memory. *Neural Comput.* 1997, 9, 1735–1780.
6. Han, S.; Kang, J.; Mao, H.; Hu, Y.; Li, X.; Li, Y.; Xie, D.; Luo, H.; Yao, S.; Wang, Y.; et al. ESE: Efficient speech recognition engine with sparse LSTM on FPGA. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 75–84.
7. Azari, E.; Vrudhula, S. An Energy-Efficient Reconfigurable LSTM Accelerator for Natural Language Processing. In Proceedings of the 2019 IEEE International Conference on Big Data (IEEE BigData 2019), Los Angeles, CA, USA, 9–12 December 2019; pp. 4450–4459.
8. Chang, A.X.M.; Martini, B.; Culurciello, E. Recurrent neural networks hardware implementation on FPGA. *arXiv* 2015, arXiv:1511.05552.
9. Guan, Y.; Yuan, Z.; Sun, G.; Cong, J. FPGA-based accelerator for long short-term memory recurrent neural networks. In Proceedings of the 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), Tokyo, Japan, 16–19 January 2017; pp. 629–634.
10. Wang, S.; Li, Z.; Ding, C.; Yuan, B.; Qiu, Q.; Wang, Y.; Liang, Y. C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 25–27 February 2018; pp. 11–20.

11. Cao, S.; Zhang, C.; Yao, Z.; Xiao, W.; Nie, L.; Zhan, D.; Liu, Y.; Wu, M.; Zhang, L. Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity. In Proceedings of the ACM/SIGDA Int’l Symp. Field-Programmable Gate Arrays, ACM, Seaside, CA, USA, 24–26 February 2019; pp. 63–72.

12. Cho, K.; Van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; Bengio, Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv 2014, arXiv:1406.1078.

13. Nurvitadhi, E.; Sim, J.; Sheffield, D.; Mishra, A.; Krishnan, S.; Marr, D. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. In Proceedings of the 2016 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, Switzerland, 29 August–2 September 2016; pp. 1–4.

14. Chen, C.; Ding, H.; Peng, H.; Zhu, H.; Ma, R.; Zhang, P.; Yan, X.; Wang, Y.; Wang, M.; Min, H.; et al. OCEAN: An on-chip incremental-learning enhanced processor with gated recurrent neural network accelerators. In Proceedings of the ESSCIRC 2017 43rd IEEE European Solid State Circuits Conference, Leuven, Belgium, 11–14 September 2017; pp. 259–262.

15. Li, S.; Wu, C.; Li, H.; Li, B.; Wang, Y.; Qiu, Q. FPGA acceleration of recurrent neural network based language model. In Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, Vancouver, BC, Canada, 2–6 May 2015; pp. 111–118.

16. Li, Z.; Ding, C.; Wang, S.; Wen, W.; Zhuo, Y.; Liu, C.; Qiu, Q.; Xu, W.; Lin, X.; Qian, X.; et al. E-RNN: Design optimization for efficient recurrent neural networks in FPGAs. In Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, 16–20 February 2019; pp. 69–80.

17. Gao, C.; Rios-Navarro, A.; Chen, X.; Liu, S.C.; Delbruck, T. EdgeDRNN: Recurrent Neural Network Accelerator for Edge Inference. IEEE J. Emerg. Sel. Top. Circuits Syst. 2020, 10, 419–432. [CrossRef]

18. Zeng, S.; Guo, K.; Fang, S.; Kang, J.; Xie, D.; Shan, Y.; Wang, Y.; Yang, H. An efficient reconfigurable framework for general purpose CNN-RNN models on FPGAs. In Proceedings of the 2018 IEEE 23rd International Conference on Digital Signal Processing (DSP), Shanghai, China, 19–21 November 2018; pp. 1–5.

19. Elman, J.L. Finding structure in time. Cogn. Sci. 1990, 14, 179–211. [CrossRef]

20. Le, Q.V.; Jaitly, N.; Hinton, G.E. A simple way to initialize recurrent networks of rectified linear units. arXiv 2015, arXiv:1504.00941.

21. Gers, F.A.; Schraudolph, N.N.; Schmidhuber, J. Learning precise timing with LSTM recurrent networks. J. Mach. Learn. Res. 2002, 3, 115–143.

22. Schuster, M.; Paliwal, K.K. Bidirectional recurrent neural networks. IEEE Trans. Signal Process. 1997, 45, 2673–2681. [CrossRef]

23. Kadetotad, D.; Berisha, V.; Chakrabarti, C.; Geo, J.S. A 8.93-TOPS/W LSTM recurrent neural network accelerator featuring hierarchical coarse-grain sparsity with all parameters stored on-chip. In Proceedings of the 2019 IEEE International Solid-State Circuits Conference (ISSCC 2019), San Francisco, CA, USA, 17–21 February 2019; pp. 119–122.

24. Intel. Stratis V Device Handbook; Intel: San Jose, CA, USA, 2020.

25. Xilinx. Xilinx Design Flow for Intel FPGA SoC Users; Xilinx: San Jose, CA, USA, 2018.

26. Marcus, M.; Santorini, B.; Marcinkiewicz, M.A. Building a large annotated corpus of English: The Penn Treebank. Comput. Linguist. 1993, 19, 313–330.

27. Kim, T.H.; Shin, J. A Resource-Efficient Inference Accelerator for Binary Convolutional Neural Networks. IEEE Trans. Circuits Syst. II Express Briefs 2020, 68, 451–455. [CrossRef]