electronics

MDPI

Article

F-LSTM: FPGA-Based Heterogeneous Computing Framework for Deploying LSTM-Based Algorithms

Bushun Liang, Siye Wang *, Yeqin Huang, Yiling Liu and Linpeng Ma

School of Artificial Intelligence, Beijing University of Posts and Telecommunications, Beijing 100876, China
* Correspondence: wsy@bupt.edu.cn

Abstract: Long Short-Term Memory (LSTM) networks have been widely used to solve sequence modeling problems. For researchers, using LSTM networks as the core and combining it with pre-processing and post-processing to build complete algorithms is a general solution for solving sequence problems. As an ideal hardware platform for LSTM network inference, Field Programmable Gate Array (FPGA) with low power consumption and low latency characteristics can accelerate the execution of algorithms. However, implementing LSTM networks on FPGA requires specialized hardware and software knowledge and optimization skills, which is a challenge for researchers. To reduce the difficulty of deploying LSTM networks on FPGAs, we propose F-LSTM, an FPGA-based framework for heterogeneous computing. With the help of F-LSTM, researchers can quickly deploy LSTM-based algorithms to heterogeneous computing platforms. FPGA in the platform will automatically take up the computation of the LSTM network in the algorithm. At the same time, the CPU will perform the pre-processing and post-processing in the algorithm.To better design the algorithm, compress the model, and deploy the algorithm, we also propose a framework based on F-LSTM. The framework also integrates Pytorch to increase usability. Experimental results on sentiment analysis tasks show that deploying algorithms to the F-LSTM hardware platform can achieve a 1.8× performance improvement and a 5.4× energy efficiency improvement compared to GPU. Experimental results also validate the need to build heterogeneous computing systems. In conclusion, our work reduces the difficulty of deploying LSTM on FPGAs while guaranteeing algorithm performance compared to traditional work.

Keywords: LSTM; FPGA; heterogeneous computing; network pruning; automation tool

Citation: Liang, B.; Wang, S.; Huang, Y.; Liu, Y.; Ma, L. F-LSTM: FPGA-Based Heterogeneous Computing Framework for Deploying LSTM-Based Algorithms. *Electronics* **2023**, *12*, 1139. https://doi.org/10.3390/electronics12051139

Academic Editor: Alexander Barkalov

Received: 13 February 2023
Revised: 23 February 2023
Accepted: 24 February 2023
Published: 26 February 2023

1. Introduction

In recent years, deep neural networks have developed rapidly. Recurrent Neural Network (RNN) [1] is a type of deep neural network used to solve sequence problems, which learns temporal information about a sequence by adding the network's output of the previous time step to the current network input. LSTM [2] networks, a particular type of RNN network, can selectively remember important information and discard unimportant information. LSTM has shown a powerful ability to learn and predict sequences and has been widely used in tasks requiring sequence modeling, such as sentiment analysis and speech recognition.

For researchers, using LSTM as a core and combining it with pre-processing and post-processing to build complete algorithms becomes a general solution when solving tasks that require sequence modeling. The solution's core is a structurally identical LSTM network. Researchers can solve various tasks by adding different pre-processing and post-processing. We refer to these solutions collectively as LSTM-based algorithms. For the raw data, we need to perform a series of pre-processing before we can input the processed sequences into the LSTM network. For example, when using LSTM to solve the sentiment analysis task [3], pre-processing in the algorithm includes tokenization and word embedding. Similarly, for the output of the LSTM network, we need to perform a series of post-processing to obtain

the final result. Again, taking sentiment analysis as an example, post-processing includes a dense layer and $softmax$ function.

The hardware platforms running deep neural networks are mainly general-purpose processors like CPU and GPU. However, as the size of LSTM networks increases, the CPU can be overwhelmed by many computations. Although GPUs are good at parallel computing, they cannot run LSTM networks efficiently due to the inherent recurrent characteristics of LSTM. Meanwhile, the high power consumption of GPU is unacceptable in some energy-sensitive applications, such as data centers and edge computing. Due to the low energy consumption, low latency, high performance, and reconfigurability of FPGA using FPGA as the hardware platform for inferring LSTM networks is an ideal solution [4].

For researchers, however, implementing an LSTM network based on FPGA requires specific software and hardware knowledge, which requires a team of domain researchers, software engineers, and hardware engineers and requires the team to spend much time designing and optimizing the implementation. For researchers, these requirements take work to meet. Moreover, a complete LSTM-based algorithm does include not only the core LSTM network but also pre-processing and post-processing. Generally speaking, the amount of computation and storage of pre-processing and post-processing is small, but the inherent logic is complex. Implementing and optimizing the operations in pre-processing and post-processing on FPGA is no less challenging than implementing an LSTM network. However, the performance improvement and energy reduction resulting from implementing pre-processing and post-processing on FPGAs are negligible for the overall algorithm.

Due to the complexity of FPGA development, the LSTM network in the famous deep neural network framework PyTorch [5] only provides support for CPU and GPU and lacks support for FPGA. Using FPGA for training the LSTM network is not a wise choice. What we need to do is to perform the inference of the LSTM network on the FPGA. It is not a challenge for researchers to use PyTorch to combine LSTM network, pre-processing, and post-processing to build a complete algorithm when solving sequence modeling tasks. What bothers researchers is how to translate the LSTM model into a high-performance, energy-efficient FPGA implementation and combine it with existing pre-processing and post-processing.

To solve these problems, we propose an FPGA-based heterogeneous computing framework, F-LSTM. With the help of F-LSTM, researchers can deploy LSTM-based algorithms into an FPGA-based heterogeneous computing platform. The FPGA in the platform will automatically assume the computation of the LSTM network in the algorithm. To ensure the algorithm's integrity, F-LSTM will leave the pre-processing and post-processing to the CPU in the heterogeneous computing platform. Meanwhile, we designed a workflow based on F-LSTM, which can help us to complete the design, compression, and deployment of LSTM-based algorithms.

The contributions of this paper are as follows.

- We proposed a heterogeneous computing framework based on FPGA, which leaves the pre-processing and post-processing to the CPU. The FPGA is responsible for the computation of LSTM to ensure the algorithm's integrity. To ensure the reliability of F-LSTM, we used the API provided by Vitis to guarantee the data transfer between the CPU and the FPGA and the correct boot of the kernel on the FPGA. To reduce the system latency, we store the LSTM model parameters in the BRAM on the FPGA, and the LSTM kernel reads them from the global memory (usually DDR on FPGA) on the FPGA before starting the computation. This read operation is performed only once during the entire life of the algorithm.
- We designed a configurable LSTM kernel using High-level synthesis language. We divided the kernel into three parts: matrix-vector multiplication, activation functions, and element-wise computation. To speed up the core matrix-vector multiplication, we prune the LSTM model, use the CRS format to store the sparse weight matrix to reduce storage, and design special matrix-vector multiplication to reduce the number

of computations. To reduce the delay caused by the exponential function term in the activation function, we use piecewise linear approximation (PWL) to implement the *sigmoid* on FPGA and replace *tanh* by *hardtanh* during network training. To reduce the overall delay of the kernel, we use HLS pragma *unroll* and *pipeline* to optimize the loop and use pragma *dataflow* to generate multiple matrix-vector multiplication PEs.
- We developed a workflow based on F-LSTM. The workflow consists of algorithm design, model compression, and algorithm deployment. This workflow can help us to complete the migration of LSTM-based algorithms from the general-purpose processor to the F-LSTM hardware platform.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 introduces the basics of LSTM, fpga-based heterogeneous computing. Section 4 describes the design of F-LSTM and the implementation of the LSTM kernel. Section 5 introduces the workflow based on F-LSTM. The experimental setup and results are described in Section 6. Section 7 concludes the paper.

2. Related Work

In recent years, there has been increased research on using FPGA to accelerate neural networks. Much FPGA-based work focuses on convolutional neural network (CNN) acceleration. Ref. [6] proposed a programmable and flexible CNN architecture, as well as data quantization strategies and compilation tools. Ref. [7] introduced an automated tool, hls4ml, for deploying ultra-low-latency, low-power deep neural networks with convolutional layers on FPGA. Ref. [8,9] implemented the high-performance CNN-based object detection algorithms on FPGA. Compared to CNN, research on accelerating RNN on FPGA, especially LSTM, is rare and homogeneous. Most of the work [4,10–16] focused on designing special pruning methods and LSTM kernels based on the methods. Ref. [4] proposed a load-balance-aware pruning method that can compress the LSTM model size by 10× with negligible loss of prediction accuracy and a hardware architecture that works directly on the sparse LSTM model. Ref. [10] proposed a structured compression technique and a comprehensive framework to optimize and automatically implement LSTM variants on FPGA. Ref. [11] proposed a structured pruning method and a hardware architecture of the compressed LSTM. Ref. [12] proposed bank-balanced sparsity(BBS) and a 3-step software-hardware co-optimization approach to applying BBS in FPGA hardware.

FPAG-based heterogeneous computing with high performance, energy efficiency, and programmability has been applied in KNN [17], Kmeans [18], and SWIFOLD [19]. Moreover, much research [20–22] has focused on running CNN on the heterogeneous computing platform.

We apply FPGA-based heterogeneous computing to LSTM-based algorithms and design a workflow that includes algorithm design, model compression, and algorithm deployment. We effectively utilize the high-performance, energy-efficient, and reconfigurability of FPGA and reduce the difficulty of deploying LSTM networks on FPGA.

3. Background

3.1. LSTM

As a particular type of RNN cell, LSTM can selectively remember some essential features and discard some unimportant features for a long time. The structure of the LSTM cell is shown in Figure 1. The LSTM cell input $[x_t, h_{t-1}]$ for the current time step t consists of the current sequence input x_t and the hidden layer output h_{t-1} from the previous time step $t-1$. Significantly, the LSTM cell contains a special memory c_t to store the cell state. It also includes an input gate, a cell gate, an output gate, and a forget gate to remember or forget features selectively. As shown in the standard LSTM cell in Figure 1, the output h_t and the cell state c_t are generated by the following equations:

$$g_t = tanh(W_g[x_t, h_{t-1}] + b_g) \quad (1)$$

$$i_t = sigmoid(W_i[x_t, h_{t-1}] + b_i) \tag{2}$$

$$o_t = sigmoid(W_o[x_t, h_{t-1}] + b_o) \tag{3}$$

$$f_t = sigmoid(W_f[x_t, h_{t-1}] + b_f) \tag{4}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{5}$$

$$h_t = tanh(c_t) \odot o_t \tag{6}$$

g_t, i_t, f_t, and o_t are the output vectors of the cell gate, input gate, forget gate, and output gate. W denotes the weight matrix (e.g., W_i is the weight matrix between the input gate and the LSTM cell input), and b denotes the bias vector (e.g., b_i is the bias vector of the input gate).

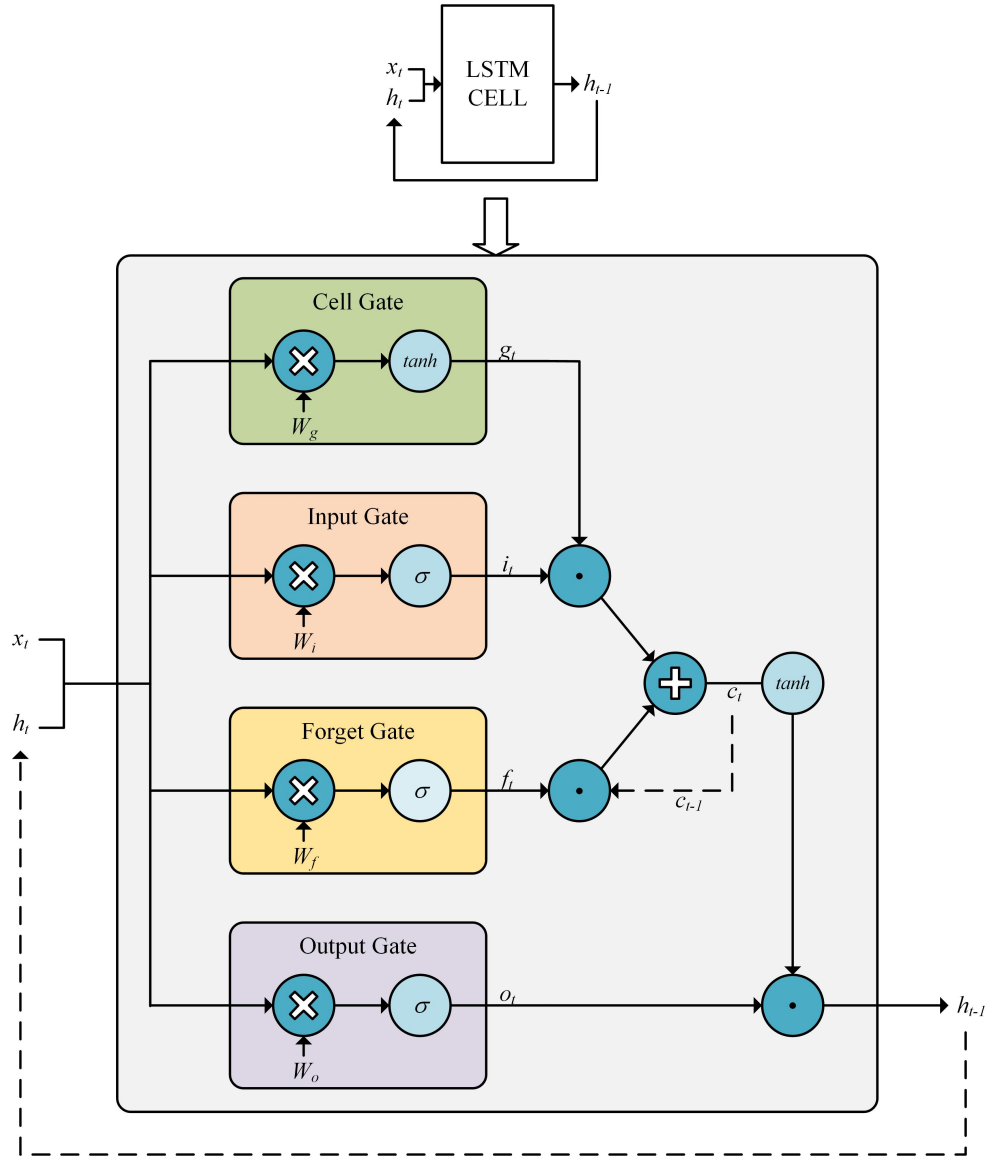

Figure 1. Structure of LSTM cell. σ denotes the *sigmoid* function, *tanh* denotes the hyperbolic tangent function, $\times$ denotes matrix-vector multiplication, $+$ operator denotes elementwise addition, and operator $\odot$ denotes elementwise multiplication.

3.2. FPGA-Based Heterogeneous Computing

FPGA-based heterogeneous computing has high performance, low power consumption, and reconfigurability features. Vitis [23] is an integrated programming environment

for FPGA acceleration supported by Xilinx. With Vitis, we can quickly build FPGA-based heterogeneous computing frameworks. Vitis' composition is shown in Figure 2. The source code is divided into host code for execution on the host and kernel code for execution on the FPGA. The host code is programmed in C/C++, which can use the XRT(Xilinx Runtime) API and can be compiled by the host compiler to obtain an executable host program. The kernel code can be programmed using RTL [24] or C/C++ HLS [25]. The hardware compiler synthesizes the kernel code into logic circuits on the FPGA and generates FPGA binaries for use by the host code. During execution, the main job of the host program is to use the APIs supported by Vitis to transfer data between the CPU and FPGA and to boot the kernel. The details of the API implementation are hidden under the XRT, driver, and shell on the FPGA. Control signals and data between the host and FPGA are transferred via PCIe.

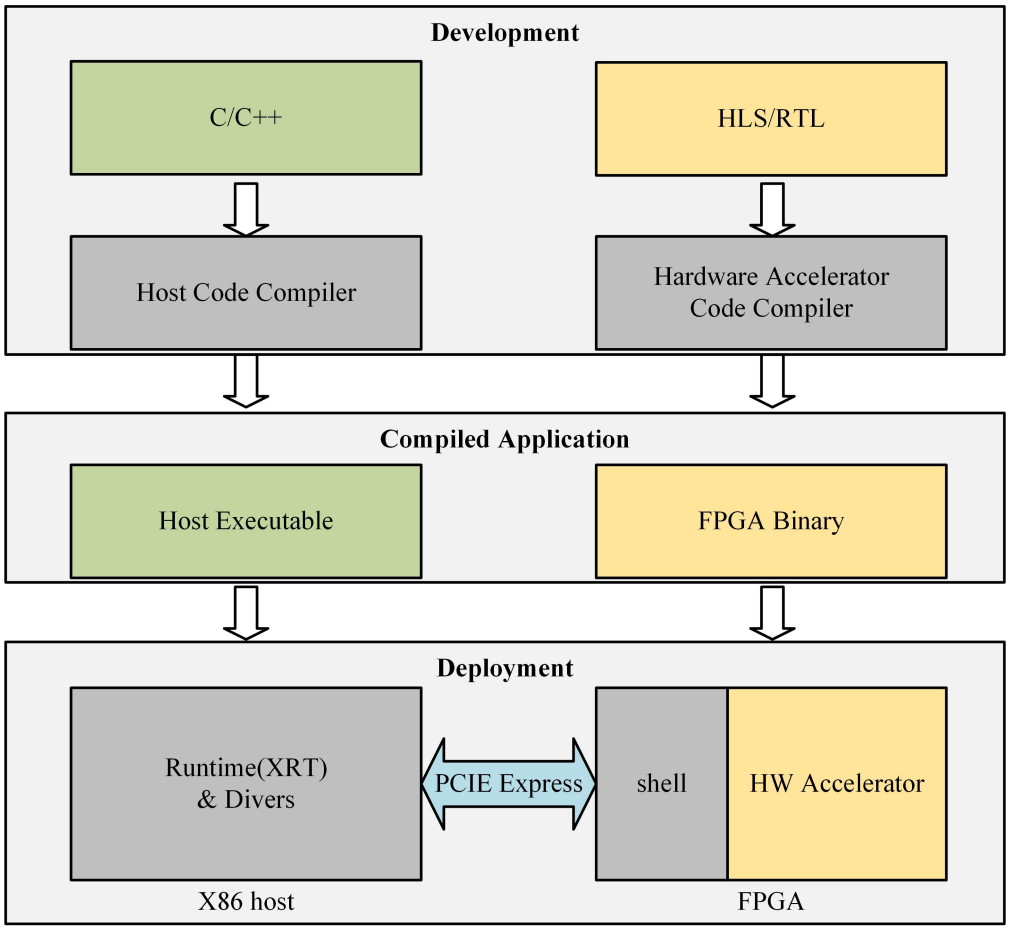

Figure 2. Vitis programming environment overview.

4. System Architecture of F-LSTM

In this paper, we propose an FPGA-based heterogeneous computing framework, F-LSTM. Figure 3 shows the system architecture of the framework. In the design, we follow the Vitis design idea. In the system we build, software and hardware work together. The host code runs on the CPU and is responsible for the pre-processing and post-processing in the LSTM-based algorithm. The kernel code runs on the FPGA and is responsible for the LSTM network inference that takes up most of the computation in the algorithm. If the FPGA is not loaded externally with the model, updating the model on the FPGA may require a re-synthesis of the project, which often takes several hours or even days. We load the model parameters from global memory at runtime to avoid this situation. Moreover, To avoid the delay caused by reading weights of the LSTM model from global memory, the kernel preloads weights into the BRAM on the FPGA before starting the computation. Without changing the model, the preloading performs only once during the whole life of

the algorithm. During the subsequent computation, the system does not need to modify the weight stored in the BRAM again. The operation of the LSTM-based algorithm on the F-LSTM platform can be divided into the following steps:

1. The host preprocesses the original data (text, audio) to obtain the feature vector sequence $[x_t, x_{t+1}, \cdots, x_{t+T}]$ (assuming the length of the sequence is T).
2. The host writes the feature vector sequence $[x_t, x_{t+1}, \cdots, x_{t+T}]$ to the global memory using the API provided by Vitis, sets the kernel parameters and boots the LSTM kernel.
3. The kernel reads the current time step input x_t from the global memory and forms the LSTM cell input with the previous time step output h_{t-1}.
4. The kernel performs LSTM network inference. Four independent processing elements(PEs) read the weights and inputs in BRAM, perform sparse matrix-vector multiplication(SPMV), and output the results to the activation function PEs. Activation function PEs compute the values of i_t, g_t, o_t, and f_t and then hand them to the element-wise PE. Element-wise PE performs element-wise multiplication, element-wise addition, and $tanh$ to obtain the output h_t and cell memory c_t.
5. The kernel writes the output h_t and cell state c_t for each time step back to the BRAM and global memory.
6. Go back to (3) and loop the operation T times.
7. The host reads the result of the LSTM network inference from the global memory using the API provided by Vitis: the output sequence $[h_t, h_{t+1}, \cdots, h_{t+T}]$ and the cell state sequence $[c_t, c_{t+1}, \cdots, c_{t+T}]$. The host selects the necessary data and performs post-processing to obtain the final result.

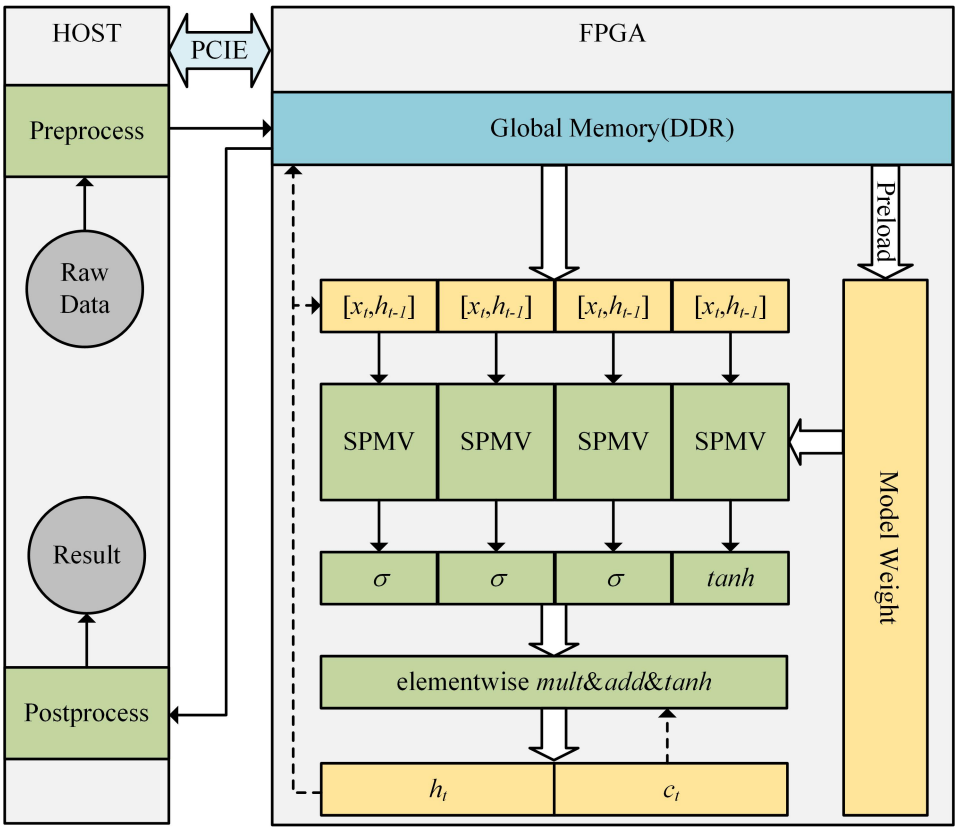

Figure 3. System architecture of F-LSTM.

The implementation details for transferring data between the kernel and the host, setting kernel parameters, and booting the kernel are all hidden under the APIs provided by Vitis. Using these APIs reduces our work and allows us to focus on implementing a high-performance LSTM kernel on an FPGA.

We divide the LSTM kernel into three parts: sparse matrix-vector multiplication, activation function, and elementwise computation. The following paper will explain each part in detail.

4.1. Sparse Matrix-Vector Multiplication

First, among all computations of LSTM, matrix-vector multiplication is the most computationally intensive operation, and reducing the computation is one way to achieve high-performance LSTM network inference. Second, storing weights directly in limited BRAMs on FPGA is impractical for large models. It is shown that trained LSTM models have much redundancy in weights, and pruning the unimportant weights (setting them to 0) only slightly or even not affects the model's accuracy but can reduce the number of weight parameters and the computational complexity. Ref. [26] provides a threshold-based weight pruning method, which prunes small weights with absolute values smaller than the threshold and retrains the remaining weights. We can apply pruning and retraining on a large model iteratively to produce a small model. The compressed LSTM network reduces computation and storage with only a slight loss of accuracy, which makes it possible to implement a high-performance LSTM network on FPGA with limited computation and storage resources. The weight pruning method converts the dense weight matrix into a sparse matrix. Since zero values are no longer involved in the computation of matrix-vector multiplication, we need to design special hardware to accelerate sparse matrix-vector multiplication.

We use the CRS [27] format to store sparse matrixes. CRS is a data structure that chooses not to store zero values of the sparse matrix, reducing the cost of storage and operations. As shown in Figure 4, CRS consists of three arrays: *Values*, *ColumnIndex* , *RowPtr*. *Values* holds the values of the non-zero weights of the sparse matrix. *ColumnIndex* and *RowPtr* encode the position information of the non-zero weights. *ColumnIndex* stores the number of columns for each weight. *RowPtr* contains the index of the first weight of each row in *Values*.

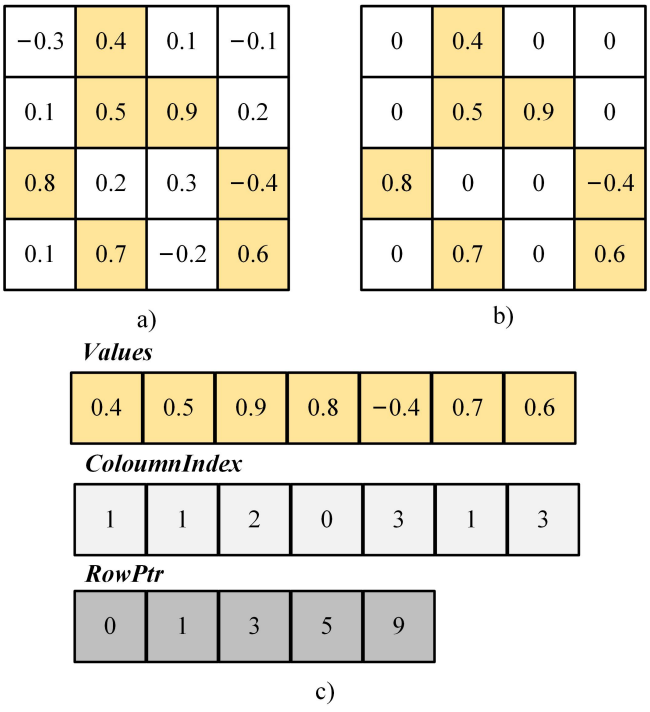

Figure 4. (**a**) dense matrix before pruning; (**b**) sparse matrix after pruning (threshold set to 0.4); (**c**) sparse matrix of CRS representation

The main code of Algorithm 1 demonstrates the computation of the matrix-vector multiplication $y = Mx$. We use the CRS to save the matrix M by $RowPtr$, $ColumnIndex$ and $Values$.The first for loop accesses each row by iteration, and the second for loop accesses each column to achieve the multiplication of the non-zero elements of the matrix M with the corresponding elements in the vector and saves the results in the vector y. To speed up the execution of the code, we use *unroll* pragma to allow iterations to occur in parallel and *pipeline* pragma to make the loop execute in a streaming style.

Algorithm 1 Sparse Matrix-Vector Multiplication.

1: **INPUT**: Vector x, Matrix M (represent by $Values$, $ColumnIndex$, $RowPtr$)
2: **for** $i = 0; i \leq NumofRows; i++$ **do**
3: $y0 = 0$
4: **for** $k = RowPtr[i]; k \leq RowPtr[i+1]; k++$ **do**
5: unroll and pipeline loop here
6: $y0 += Values[k] * x[ColumnIndex[k]]$
7: **end for**
8: $y[i] = y0$
9: **end for**
10: **RETURN** y

The input of SPMV PE consists of the input x_t for the current time step and the output vector h_{t-1} for the previous time step. The input vector multiplies with four independent weight matrixes to obtain four intermediate vectors. To reduce the latency of the kernel, we use the $dataflow$ pragma to generate four independent PEs, each performing matrix-vector multiplication in parallel. All four PEs need to read the input of the LSTM cell, which may lead to memory access conflicts and, thus, higher latency for the whole kernel. To avoid memory conflict access, we must copy four copies of the input vector before performing sparse matrix-vector multiplication.

4.2. Activation Function

The inference of LSTM includes the computation of nonlinear activation functions *tanh* and *sigmoid*. Exponential terms appear in the activation function's computation, making it very difficult to implement the activation function directly in FPGA. The taylor series approximation [28], CORDIC algorithm [29], is a commonly used alternative. The Taylor series approximation method converts the activation function into a simple polynomial calculation by solving the Taylor expansion of the activation function, then removing the higher order terms and keeping only the lower order terms needed to guarantee accuracy. Polynomial computation requires Digital signal processors (DSP). Even though DSPs have been integrated into modern FPGAs, for neural networks, the limited DSP resources should be used for the large number of parallel computations in the network. The CORDIC algorithm is an iterative algorithm that allows for higher accuracy but requires more FPGA resources.

We implement *sigmoid* in FPGA using PWL, which requires fewer FPGA resources. The basic idea of PWL is to use a series of functions to approximate a nonlinear function. On the FPGA, the breakpoints of these functions can be stored in a lookup table, and the intermediate values are then computed based on these functions. On FPGA, if one of the inputs to the multiplier is a power of 2, then the multiplication operation can be replaced by a shift. We use Equation (7) [30] to approximate *sigmoid*, which requires only a tiny amount of FPGA resources.

$$f(x) = \begin{cases} 1 - \frac{1}{2}\left(1 - \frac{1}{4}|x|\right)^2, & 0 < x \leq 4 \\ \frac{1}{2}\left(1 - \frac{1}{4}|x|\right)^2, & -4 < x \leq 0 \\ 1, & 4 < x \\ 0, & x \leq -4 \end{cases} \tag{7}$$

Since *tanh* is challenging to approximate using PWL, we choose to replace *tanh* in LSTM with *hardtanh* in the Equation (8) when building the algorithm. Experimental results on sentiment analysis in Figure 5 demonstrate that this approach does not increase the training time or decrease the model's final accuracy.

$$hardtanh(x) = \begin{cases} -1, & x < -1 \\ x, & -1 \leq x \leq 1 \\ 1, & x > 1 \end{cases} \tag{8}$$

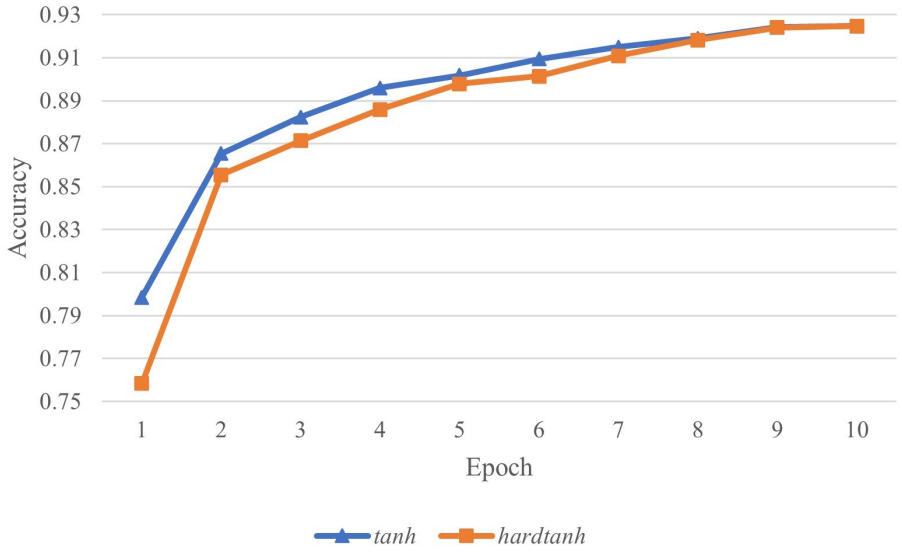

Figure 5. Comparison between *tanh* and *hardtanh* in terms of training time and model's accuracy.

4.3. Elementwise Computation

As shown in Figure 1, after performing matrix-vector multiplication and activation function, we can get the output vectors of the input gate, output gate, cell gate, and forget gate. As described in Equations (5) and (6), four vectors are involved in elementwise multiplication, elementwise addition, and *tanh* to obtain the cell state vector c_t and the output vector h_t.

5. WorkFlow

We can transform LSTM-based algorithms running on general-purpose processors to algorithms running on the F-LSTM platform. Figure 6 shows the workflow of algorithm transformation. First, when designing the algorithm, it is necessary to divide the algorithm into three parts: pre-processing, LSTM network, and post-processing. The pre-processing and post-processing are written using Python code, and PyTorch provides the LSTM network. The LSTM network can be trained to obtain an accurate model. To reduce the amount of storage and computation, we also need to compress the model, which involves iterative pruning and fine-tuning. Ultimately, we will obtain a tiny LSTM model with little accuracy loss. The compressed LSTM model comprises a complete network structure and sparse weight matrix. The input size and the size dimension describe the network structure (we do not consider multilayer LSTM networks for now), which we call configuration. The weights are the sparse matrixes and bias vectors of the gates. We need to transform the

sparse matrix into three arrays of CRS and generate a binary file for the LSTM kernel to preload.

With the configuration of the LSTM model and a pre-prepared Vitis project template, we can transform the LSTM network into a project that can be synthesized by Vitis, containing both the host code and the kernel code. We use HLS to program the kernel. Although it may result in slightly worse optimal performance than RTL-based designs, HLS-based designs increase the level of abstraction, reduce iteration time, simplify the verification phase, and allow for more exploration and evaluation of the design solution. Most importantly, generating HLS code is much simpler than generating RTL code. We implement the *forward* interface in the host code. Primary functions of the *forward* interface include:

1. transferring the vector sequence into global memory for preloading by the LSTM kernel
2. setting the kernel parameters and booting the kernel
3. transferring the outputs of the kernel back to the host

The role of our *forward* interface is similar to *forward* interface in Pytorch, which provides an API with hidden details for the user to call. To enable better migration of the PyTorch project, we use *ctype* to bind the *forward* interface in the host code so that researchers can call it using the Python language. With *forward* interface, researchers can easily reuse previous pre-processing and post-processing code to reconstruct the complete LSTM-based algorithm while using the generated LSTM kernel to accelerate network inference.

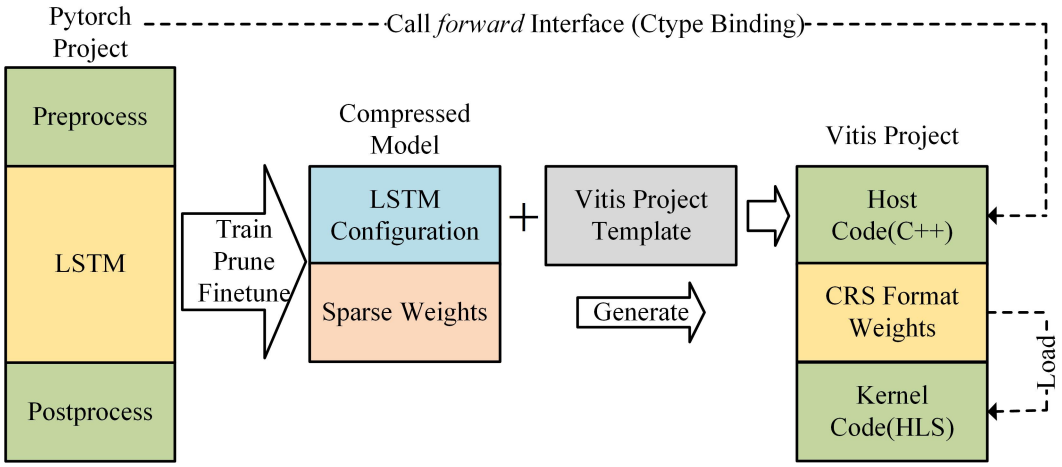

Figure 6. Workflow based on F-LSTM, which consists of three steps: designing the algorithm, compressing the model, and deploying the algorithm.

6. Experimental Results

6.1. Experimental Setup

The F-LSTM hardware platform consists of a host platform and an FPGA platform: (1) The host platform is an Intel(R) Xeon(R) D-2183IT, which runs at 2.2 GHz. (2) The FPGA platform is a KCU1500 accelerated development board with 16GB DDR4 external memory. The maximum number of BRAM, DSP48E, FF, and LUTs available on the KCU1500 are 2160, 2760, 663,360, and 331,680, respectively. The CPU and FPGA are connected via a PCIe 2.0 × 8 interface. In addition, the hardware used to compare with the heterogeneous computing platform is the Nvidia Geforce GTX 1080Ti.

Our experimental data are obtained from the IMDB dataset [31], a movie review dataset for sentiment analysis. The IMDB dataset contains a training set of 25,000 and a test set of 25,000. Movie reviews are labeled as positive and negative based on the sentiment of the text. We use PyTorch 1.6 to implement the LSTM-based sentiment analysis algorithm and train through the IMDB dataset to obtain an accurate LSTM model. The input size of the LSTM network is 300, the hidden size is 1500, and we set the batch size to 1 to meet the

real-time application. We use iterative pruning and fine-tuning to compress the model until the sparsity reaches 80% (sparsity refers to the ratio of the pruned weights to the original weights). We use Vitis 2021.1 to design the LSTM template project and ctype 3.1 as a python binding tool. The sentiment analysis algorithm in F-LSTM is shown in Figure 7, where the tokenization, embedding, dense layer, and $softmax$ functions are deployed on the CPU while the LSTM network runs on the FPGA.

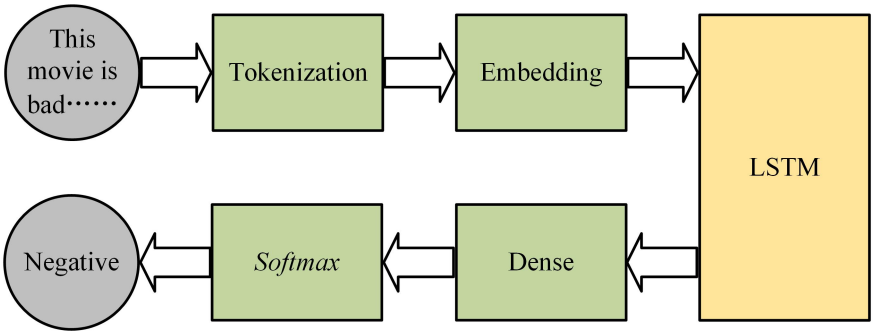

Figure 7. LSTM-based algorithm for sentiment analysis.

6.2. Resource Utilization and Performance

First, we measured the resource utilization and performance of our FPGA implementation. In the experiments, performance can be measured by latency, which in this paper refers to the execution time required to predict all reviews in the test set. We also compared the two schemas' resource utilization and performance (generating a single sparse matrix-vector multiplication PE or four sparse matrix-vector multiplication PEs). We performed a full resource report on the LSTM kernel using the Vitis tool. When the target frequency is set to 300 MHz, the resource utilization is shown in Table 1. It can be seen that although the scheme of generating more PEs will take up more system resources, the concurrency of the system will also be higher, and the latency of the algorithm will be lower.

Table 1. Comparison between two schema (generating one SPMV PE or four SPMV PEs).

SPMV NUM	BRAM	DSP48	FF	LUT	Latency (s)
1	987 (46%)	220 (8%)	208,980 (31.5%)	109,921 (33.1%)	2986
4	1072 (50%)	966 (35%)	215,897 (32.5%)	121,535 (36.7%)	752

6.3. Compare with GPU

Our most important goal is to optimize latency and energy efficiency. The lower the latency or higher the energy efficiency, the better the performance of the FPGA design. Energy efficiency is the use of less energy to perform the same task or produce the same result. In our experiment, energy efficiency can be calculated by multiplying the power consumption by the latency. We use the power analysis feature of VivadoTM to estimate FPGA power consumption and the NVIDIA System Management Interface (nvidiasmi) command line tool to measure GPU power consumption. Since both platforms use the CPU for some additional work, we do not add the CPU power consumption to the overall power consumption. Both the GPU and the FPGA run pruned models. Figure 8 shows the power consumption and latency of the FPGA and GPU. It can be seen that the FPGA achieves a 1.8× speedup and 5.4× energy efficiency compared to the GPU.

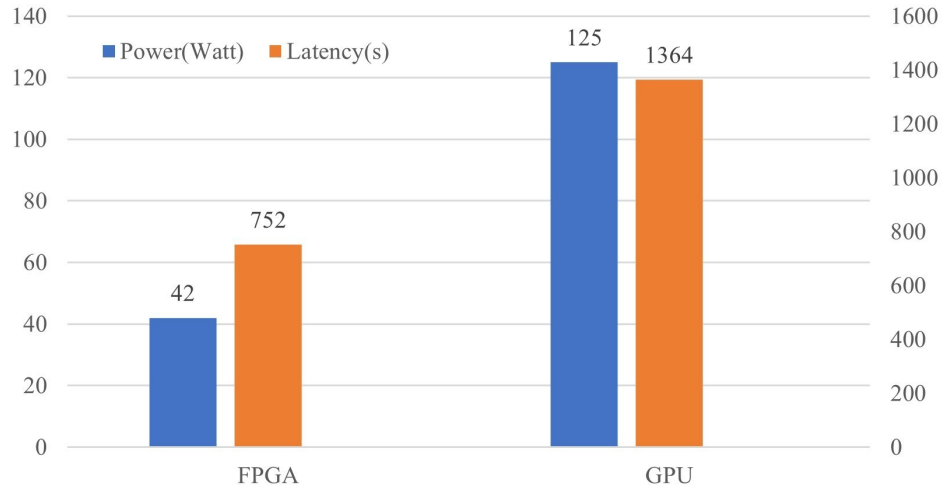

Figure 8. Power and Latency on FPGA and GPU.

Table 2 shows the size and accuracy of the model running on FPGA and GPU. To illustrate the effect of pruning on the accuracy, we also measured the size and accuracy of the original dense model. We measure the size of the LSTM model running on the GPU through Pytorch's API. The size of the LSTM model running on FPGA refers to the size of the binary file used for FPGA preloading. The accuracy is the ratio of the number of correct predictions to the total number of input samples. As we can see, the pruning method can significantly reduce the model size with almost no impact on the model's accuracy. Even with a sparsity as high as 80%, the accuracy of the pruned model running on the GPU does not degrade, and the accuracy of the pruned model running on the FPGA, although degraded, is negligible (the degradation in accuracy may be caused by the implementation of the *sigmoid* function using PWL). Moreover, using the self-CRS representation of the sparse weight matrix can further reduce the model size since we only need to store a small number of weights.

Table 2. Size and Accuracy of the model running on FPGA and GPU.

Model	Sparsity	Platform	Size (MB)	Accuracy
Pruned Model (CRS)	80%	FPGA	10.42	91.5%
Pruned Model	80%	GPU	21.53	92.4%
Dense Model	0%	GPU	57.82	92.3%

6.4. Necessity of Heterogeneous Computing

To evaluate the necessity of using heterogeneous computation, we use the system API to measure the latency of each layer in pre-processing, LSTM network, and post-processing. In the algorithm for sentiment analysis, the pre-processing includes the tokenization and embedding, and the post-processing includes the dense layer and $softmax$ function. From Table 3, we can see that the latency of pre-processing and post-processing is only a tiny percentage of the latency of the whole algorithm. All we need to do is accelerate LSTM network inference. Moreover, runtime preloading the model parameters from external storage is necessary. We measured the resources and time required to update the model parameters for both ways of updating them. As shown in Table 4, it can be seen that removing the preloading logic saves a few resources, while the time required for resynthesizing the project is much longer than the preload operation.

Table 3. Latency of layers in the sentiment analysis algorithm.

Latency(s) Platform / Layer	F-LSTM	GPU
Tokenization	2.43	2.43
Embedding	11.28	10.91
LSTM	752.3	1364.3
Dense	15.21	10.02
Softmax	5.92	6.91
Total	787.14	1394.93

Table 4. Comparison of two methods of updating model weights.

Method	BRAM	DSP48	FF	LUT	Time Required
Synthesis Project	1004	966	201,330	102,321	14.1 s
Preload	1072	966	215,897	121,535	7 h 32 m

6.5. Compare with Other Works

Table 5 shows how our work compares to others regarding performance, usability, and scalability. For a better performance comparison, the throughput has been normalized to 300 MHz, assuming the throughput depends linearly on the frequency. As can be seen, our work has a significant performance advantage. Runtime loading refers to the implementation providing an interface for updating the model weights on the FPGA, which means that we can spend very little time updating new models with higher accuracy onto the FPGA instead of spending hours or even days resynthesizing the project. Finally, only our work extends the scalability of the LSTM implementation; none of the other implementations integrate with currently popular deep learning frameworks.

Table 5. Comparison of implementations for LSTM network on FPGA.

Works	FPGA	Freq (MHz)	Perform (GOP/s)	Runtime Loading	Integrated
[13]	XCZU6EG	238	9.63	Supported	-
[14]	XC7VX485	142	0.95	Supported	-
[15]	XC7Z045	150	14.52	Supported	-
[16]	XC7Z020	140	9.71	Not supported	-
This Work	KCU1500	300	43.1	Supported	PyTorch

7. Conclusions

We present an FPGA-based heterogeneous computing framework, F-LSTM, and a workflow based on F-LSTM. Guided by the workflow, researchers can design, compress, and deploy LSTM-based algorithms to the F-LSTM hardware platform to meet the requirements of high performance and energy efficiency. The workflow integrated with Pytorch significantly reduces the difficulty of deploying LSTM-based algorithms on FPGA-based heterogeneous computing platforms. We designed and optimized the LSTM kernel for the pruning model to better utilize the FPGA. Experimental results show that running the algorithm on the F-LSTM hardware platform can outperform the GPU in terms of speed and energy efficiency with little degradation in accuracy. Our work also validates the need to build a heterogeneous computing platform. Compared to some other work, we have

a performance advantage and perform better in terms of ease of use and scalability. For future work, we intend to integrate TensorFlow into our framework and add compression methods such as knowledge distillation and quantization to our workflow. In conclusion, our work reduces the difficulty of deploying LSTM models on FPGAs while guaranteeing the performance of LSTM-based algorithms.

Author Contributions: Conceptualization, B.L. and S.W.; methodology, B.L.; software, B.L.; validation, B.L. and Y.H.; formal analysis, B.L. and Y.L.; investigation, B.L.; resources, Y.L.; data curation, B.L.; writing—original draft preparation, B.L.; writing—review and editing, S.W.; visualization, L.M.; supervision, Y.L.; project administration, S.W.; funding acquisition, S.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by National Natural Science Foundation of China under Grant No. 61971062.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: This work is conducted on the platform of the Center for Data Science of Beijing University of Posts and Telecommunications.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zaremba, W.; Sutskever, I.; Vinyals, O. Recurrent neural network regularization. *arXiv* **2014**, arXiv:1409.2329.
2. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef] [PubMed]
3. Li, D.; Qian, J. Text sentiment analysis based on long short-term memory. In Proceedings of the 2016 First IEEE International Conference on Computer Communication and the Internet (ICCCI), Wuhan, China, 13–15 October 2016; pp. 471–475.
4. 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, New York, NY, USA, 22–24 February 2017; pp. 75–84.
5. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. Pytorch: An imperative style, high-performance deep learning library. In Proceedings of the Advances in Neural Information Processing Systems 32 (NeurIPS 2019), Vancouver, BC, Canada, 8–14 December 2019.
6. Guo, K.; Sui, L.; Qiu, J.; Yu, J.; Wang, J.; Yao, S.; Han, S.; Wang, Y.; Yang, H. Angel-eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2017**, *37*, 35–47. [CrossRef]
7. Aarrestad, T.; Loncar, V.; Ghielmetti, N.; Pierini, M.; Summers, S.; Ngadiuba, J.; Petersson, C.; Linander, H.; Iiyama, Y.; Di Guglielmo, G.; et al. Fast convolutional neural networks on FPGAs with hls4ml. *Mach. Learn. Sci. Technol.* **2021**, *2*, 045015. [CrossRef]
8. Nguyen, D.T.; Nguyen, T.N.; Kim, H.; Lee, H.J. A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2019**, *27*, 1861–1873. [CrossRef]
9. Zhao, R.; Niu, X.; Wu, Y.; Luk, W.; Liu, Q. Optimizing CNN-based object detection algorithms on embedded FPGA platforms. In Proceedings of the International Symposium on Applied Reconfigurable Computing, Delft, The Netherlands, 3 April 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 255–267.
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, New York, NY, USA, 25–27 February 2018; pp. 11–20.
11. Wang, S.; Lin, P.; Hu, R.; Wang, H.; He, J.; Huang, Q.; Chang, S. Acceleration of LSTM with structured pruning method on FPGA. *IEEE Access* **2019**, *7*, 62930–62937. [CrossRef]
12. 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 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, New York, NY, USA, 24–26 February 2019; pp. 63–72.
13. Chen, K.; Huang, L.; Li, M.; Zeng, X.; Fan, Y. A compact and configurable long short-term memory neural network hardware architecture. In Proceedings of the 2018 25th IEEE International Conference on Image Processing (ICIP), Athens, Greece, 7–10 October 2018; pp. 4168–4172.
14. 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), Chiba, Japan, 16–19 January 2017; pp. 629–634.

15. Chang, A.X.M.; Culurciello, E. Hardware accelerators for recurrent neural networks on FPGA. In Proceedings of the 2017 IEEE International Symposium on Circuits and Systems (ISCAS), Baltimore, MD, USA, 28–31 May 2017; pp. 1–4.
16. Ferreira, J.C.; Fonseca, J. An FPGA implementation of a long short-term memory neural network. In Proceedings of the 2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Cancun, Mexico, 30 November–2 December 2016; pp. 1–8.
17. Pu, Y.; Peng, J.; Huang, L.; Chen, J. An efficient knn algorithm implemented on fpga based heterogeneous computing system using opencl. In Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, Vancouver, BC, Canada, 2–6 May 2015; pp. 167–170.
18. Tang, Q.Y.; Khalid, M.A. Acceleration of k-means algorithm using altera sdk for opencl. *ACM Trans. Reconfig. Technol. Syst.* **2016**, *10*, 1–19. [CrossRef]
19. Rucci, E.; Garcia, C.; Botella, G.; De Giusti, A.; Naiouf, M.; Prieto-Matias, M. SWIFOLD: Smith-Waterman implementation on FPGA with OpenCL for long DNA sequences. *BMC Syst. Biol.* **2018**, *12*, 43–53. [CrossRef] [PubMed]
20. Spagnolo, F.; Perri, S.; Frustaci, F.; Corsonello, P. Energy-efficient architecture for CNNs inference on heterogeneous FPGA. *J. Low Power Electron. Appl.* **2019**, *10*, 1. [CrossRef]
21. Zhang, J.; Li, J. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, New York, NY, USA, 22–24 February 2017; pp. 25–34.
22. Suda, N.; Chandra, V.; Dasika, G.; Mohanty, A.; Ma, Y.; Vrudhula, S.; Seo, J.S.; Cao, Y. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, New York, NY, USA, 21–23 February 2016; pp. 16–25.
23. Kathail, V. Xilinx vitis unified software platform. In Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, New York, NY, USA, 23–25 February 2020; pp. 173–174.
24. Chu, P.P. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*; John Wiley & Sons: Hoboken, NJ, USA, 2006.
25. Gajski, D.D.; Dutt, N.D.; Wu, A.C.; Lin, S.Y. *High—Level Synthesis: Introduction to Chip and System Design*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2012.
26. Han, S.; Mao, H.; Dally, W.J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv* **2015**, arXiv:1510.00149.
27. Smailbegovic, F.; Gaydadjiev, G.N.; Vassiliadis, S. Sparse matrix storage format. In Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing, Utrecht, The Netherlands, 11–18 November 2005; pp. 445–448.
28. Temurtas, F.; Gulbag, A.; Yumusak, N. A study on neural networks using Taylor series expansion of sigmoid activation function. In Proceedings of the International Conference on Computational Science and Its Applications, Assisi, Italy, 14–17 May 2004; Springer: Berlin/Heidelberg, Germany, 2004; pp. 389–397.
29. Chen, H.; Jiang, L.; Luo, Y.; Lu, Z.; Fu, Y.; Li, L.; Yu, Z. A CORDIC-based architecture with adjustable precision and flexible scalability to implement sigmoid and tanh functions. In Proceedings of the 2020 IEEE International Symposium on Circuits and Systems (ISCAS), Seville, Spain, 12–14 October 2020; pp. 1–5.
30. Ngah, S.; Bakar, R.B.A. Sigmoid Function Implementation Using the Unequal Segmentation of Differential Lookup Table and Second Order Nonlinear Function. *J. Telecommun. Electron. Comput. Eng.* **2017**, *9*, 103–108.
31. Maas, A.L.; Daly, R.E.; Pham, P.T.; Huang, D.; Ng, A.Y.; Potts, C. Learning Word Vectors for Sentiment Analysis. In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Portland, OR, USA, 19–24 June 2011; pp. 142–150.