



# Multi-layer LSTM Parallel Optimization Based on Hardware and Software Cooperation

Qingfeng Chen, Jing Wu<sup>(✉)</sup>, Feihu Huang, Yu Han, and Qiming Zhao

School of Computer Science and Technology, Wuhan University of Science and  
Technology, Wuhan 430065, China

{chqhong,wujingecs,huangfeihu,kenlig}@wust.edu.cn

**Abstract.** LSTM's special gate structure and memory unit make it suitable for solving problems that are related to time series. It has excellent performance in the fields of machine translation and reasoning. However, LSTM also has some shortcomings, such as low parallelism, which leads to insufficient computing speed. Some existing optimization ideas only focus on one of the software and hardware. The former mostly focuses on model accuracy, and CPU accelerated LSTM doesn't dynamically adjust to network characteristics; While the latter can be based on the LSTM model structure. Customized accelerators are often limited by the structure of LSTM and cannot fully utilize the advantages of the hardware. This paper proposed a multi-layer LSTM optimization scheme based on the idea of software and hardware collaboration. We used the pruning by row scheme to greatly reduce the number of parameters while ensuring accuracy, making it adapt to the parallel structure of the hardware. From the perspective of software, the multi-layer LSTM module was analyzed. It was concluded that some neurons in different layers could be calculated in parallel. Therefore, this paper redesigned the computational order of the multilayer LSTM so that the model guaranteed its own timing properly and it was hardware friendly at the same time. Experiments showed that our throughput increased by 10x compared with the CPU implementation. Compared with other hardware accelerators, the throughput increased by 1.2x-1.4x, and the latency and resource utilization had also been improved.

**Keywords:** LSTM · Software and hardware cooperation · Parallelism · RNN · NLP

## 1 Introduction

With the fast advances in Information technologies [1–3] and big data algorithms [4–6], NLP (*natural language processing*) is a fast growing area with the powerful *recurrent neural network* (RNN) [7] that is more sensitive to temporal tasks. However, the problem of gradient explosion or gradient disappearance occurs

when the neural network is too deep or has too much temporal order. So, RNN can learn only short-term dependencies. To learn long-term dependencies, the long short-term memory network LSTM [8] emerged based on RNNs. LSTM uses unique gate structure to avoid RNN-like problems by forgetting some information. LSTM has a wide range of applications in NLP, especially in the field of machine translation, where it is much more effective than RNN. Although the special “gate” of LSTM solves the problem that RNN cannot learn long-term dependencies, it also strengthens the data dependency between modules, leading to its high requirement for temporality. It causes LSTM’s low parallelism.

Many optimizations for LSTMs have also emerged. Some optimization models [9–11] tended to optimize for model accuracy, and most of them ignored model training speed and model size. There were also some optimizations to accelerate the inference speed of the model by hardware accelerators [12–14]. FPGA became the primary choice for hardware-accelerated LSTMs due to their excellent performance in accelerating CNNs. [15] accelerated the LSTM model characteristics by designing custom accelerators. By analyzing the LSTM model, implementing each module in hardware, and designing custom hardware structure to complete the acceleration according to the model characteristics.

Based on the idea of hardware-software collaborative optimization, this paper presents a modular analysis and collaborative design of LSTM. By implementing pruning, quantization, and improved data flow, the network optimization is carried out in collaboration with hardware and software to maximize parallelization while ensuring normal timing and no data dependencies. The contributions of this article are:

- 1) The pruning method in this paper uses pruning by row, by which the final pruning result can ensure the same number of elements in each row and prepare for the parallelism design of the hardware.
- 2) In this paper, we propose a method for optimizing neural networks based on collaborative ideas of hardware and software, and a specific design on a multilayer LSTM model.
- 3) The modularity analysis of LSTM from a software perspective concludes that it is possible to compute some cells in different layers in parallel. Based on this, the computational order of the multilayer LSTM is redesigned in this paper so that the model is hardware-friendly while properly ensuring its timing.

The rest of the paper is organised as follows: Sect. 2 describes the related work; Sect. 3 introduces the idea of software and hardware co-optimization of the LSTM and the implementation details in software; Sect. 4 describes the design of the corresponding hardware structure based on the optimization effect on the software; The results and analysis of the experiments are presented in Sect. 5. Section 6 concludes this paper.

## 2 Related Work

Machine learning [16, 17] have been widely applied in various areas and applications, such as finance [18, 19], transportation [20], and tele-health industries [21]. The optimization problem [22, 23] of neural networks is top-rated, and it

is divided into two main directions including, optimizing network models and designing hardware accelerators.

The design of hardware accelerators can be divided into two main categories. One category is for data transmission. Three types of accelerators were designed by [14] for different data communication methods. The first one was to stream all data from off-chip memory to the coprocessor. This had high performance but was limited by the off-chip memory bandwidth. The second one used on-chip memory to store all the necessary data internally. This scheme achieved a low off-chip memory bandwidth, but it is limited by the available on-chip memory. The third one balanced the former two options to achieve high performance and scalability.

The other category focused on computational processes. Paper [24] proposed a hardware architecture for LSTM neural networks that aimed to outperform software implementations by exploiting their inherent parallelism. Networks of different sizes and platforms were synthesized. The final synthesized network ran on an i7-3770k desktop computer and outperformed the custom software network by a factor of 251. Paper [25] proposed an fpga-based LSTM-RNN accelerator that flattened the computation inside the gates of each LSTM, and the input vectors and weight matrices were flattened together accordingly. Execution was pipelined between LSTM cell blocks to maximize throughput. A linear approximation was used to fit the activation function at a cost of 0.63% error rate, significantly reducing the hardware resource consumption.

Optimizing network models and designing hardware accelerators could have the effect of optimizing the network, but they only considered one optimization direction. The former only considered the optimization of the network and ignores the reconfigurable design of hardware. The latter focused on hardware design without considering the characteristics of the network itself, and there was no synergistic design between software and hardware.

### 3 LSTM Software and Hardware Co-optimization

Our design is based on both of the software dimension and the hardware dimension, considering how to design to make it hardware friendly when optimizing the algorithm. And when designing the hardware architecture, consider how to adapt the hardware architecture to better support the software algorithm. In this paper, we propose the idea of software and hardware co-optimization. It is a spiral optimization method that considers the software optimization method while adjusting the software optimization and hardware structure with feedback according to the characteristics of the target hardware structure. The software optimization is combined with the hardware design to design the hardware structure supporting the above model from the hardware perspective while the structure of the neural network model and the number of parameters are reasonably designed.

#### 3.1 Approach

In this paper, we discuss on that neural network algorithms and hardware architectures interact with each other in the computational process with a

considerable degree of collaboration. Therefore, software optimization needs to combine with hardware optimization. Often, the number of model parameters is too large for the hardware, and it is necessary to determine whether direct deployment to the hardware is feasible before performing the analysis. If the model parameters put too much pressure on the computational resources of the hardware, the number of model parameters can be reduced using pruning. There are various pruning methods, and while reducing the number of parameters, it is necessary to choose a pruning method that is more compatible with hardware storage and computation. If the transfer pressure of data in each computing module is too high, the data can be quantized to relieve the transfer pressure.

After compressing the model, we need to analyze the model as a whole. From a software perspective, the algorithms and data flow between different modules are analyzed. The relationship between the modules also needs to be sorted. From the hardware perspective, the data dependencies of each module of the model are analyzed, and which parts can be optimized by parallelization, data flow design, and pipeline design are considered. It is worth noting that these two tasks are carried out at the same time. While considering algorithm optimization and tuning the model structure, one needs to consider how to tune it to make it hardware-friendly. While considering hardware optimization, one needs to consider how to tune the algorithm and structure to make it decoupled and design hardware structures with higher parallelism and lower latency. Finally, the model structure and hardware architecture have good adaptability through the collaborative design of hardware and software.

### 3.2 Model Compression

With the development of neural networks, the accuracy of model is no longer the only metric to evaluate a model, and researchers are increasingly focusing on other metrics of the model in the application domain, such as model size, model consumption of resources, etc. [26]. While pruning and quantization as the common methods for model compression, their algorithms are being optimized as the demand expands. [27] proposed a deep compression by designing a three-stage pipeline: pruning, trained quantization and Huffman coding. Combining multiple compression methods onto a neural network, it was finally demonstrated experimentally that the requirements of the model were reduced by a factor of  $35\times$  to  $49\times$  without the accuracy loss.

**Pruning:** Network pruning is one of the commonly used model compression algorithms. In the pruning process, we can construct redundant weights and keep important weights according to certain criteria to maintain maximum accuracy. There are many pruning methods, and they each have their focus. Since the computation process of LSTM contains a large number of matrix vector multiplication operations, this paper focuses on the pruning operation of the weight matrix of LSTM. The pruning method in this paper is the less common pruning by row, which differs from the general pruning by threshold method in that the selection of pruning range by row is changed. It takes a row of the weight matrix

as a whole and selects a threshold for pruning according to the percentage of a row, and by this way, the final pruning result can ensure a consistent number of elements in each row. In the hardware implementation of matrix operations for high parallelism in computation, we expand the matrix by rows and compute it in parallel from row to row. The idea of hardware-software collaboration is fully reflected here. The accuracy of the model was degraded after pruning, so retraining was performed after pruning as a way to improve the accuracy. It is worth noting that the pruned parameters can easily be added with an offset value, which will cause the pruning effect to be lost without doing something during the retraining process. Therefore, we add a mask matrix during retraining to ensure that the 0 elements of the weight matrix are not involved in training before the weight matrix is involved during retraining. The shape of the mask matrix is the same as that of the weight matrix. When the element value of the weight matrix is 0, the element value of the mask matrix for the position is also 0.

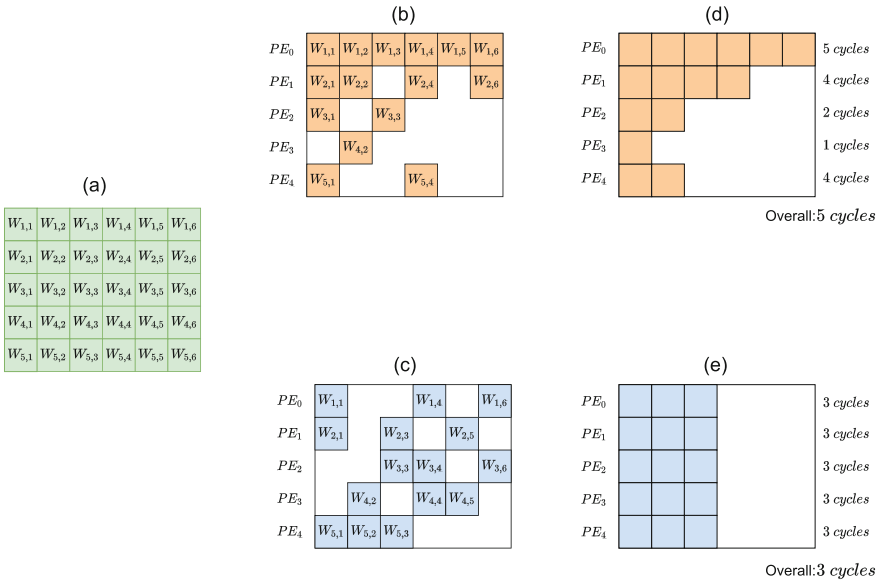


Fig. 1. Pruning by row and its benefit for parallel processing.

Figure 1(a) shows the matrix of memory gate  $O_i$ 's weights in a certain neuron. Figure 1(b) shows the result map  $O_i^a$  after unbalanced pruning while Fig. 1(c) is the result map  $O_i^b$  after pruning by row. After pruning the weight matrix, we assign each row of the matrix to different PEs for parallel computation. Figure 1(d) and Fig. 1(e) are the running times of several PEs after the two pruning, respectively, and we can see that the unbalanced pruning Fig. 1(d) will result in a different amount of data per row, which leads to a large difference in the elapsed time between the parallel PEs.

The total elapsed time will be determined by the computation time of the most loaded PEs, while the other PEs will have different degrees of waiting, which leads to a decrease in running efficiency. The total time is five clocks. While pruning by row makes each row have the same amount of data, parallel PEs do not experience waiting, making the overall operation efficiency guaranteed to be high. With the guarantee that all rows require the same clock cycle, the total time is only three clock cycles.

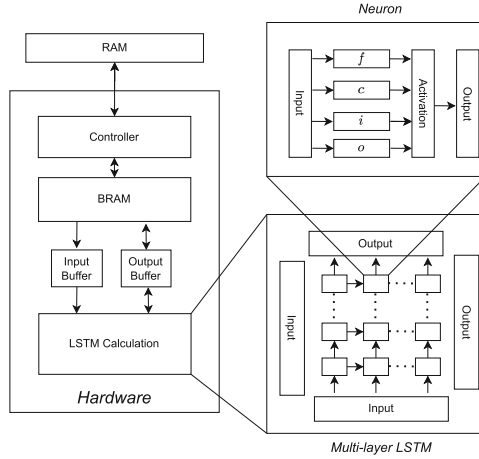
**Quantification:** Quantification is another universal approach to model compression. In general, when a well-trained model infers, the model can deal with some input noise. It means that the model can ignore all non-essential differences between the inference and training samples. To some extent, low precision can be considered as a source of noise that provides non-essential differences. Therefore, theoretically, the neural network can give accurate results even if the data precision is low. At the same time, neural networks may occupy a large amount of storage space. This means that the network requires not only a large amount of memory but also a large number of computational resources during operation. This problem also provides the need for quantification.

The parameters of the LSTM weight matrix are 32-bit single-precision floating-point numbers. The memory requirement is very large when the network size is large, and the computation of floating-point numbers requires a lot of hardware resources. [28] used a linear quantification strategy for both weights and activation functions. The authors analyzed the dynamic weight range of all matrices in each LSTM layer and explore the effect of different quantification bits on the LSTM model. In this paper, we quantify float-32 to fixed-8. Although a certain amount of accuracy is lost, multiple data can be read at a time for processing, which substantially improves the efficiency of computing.

By compressing the model, we greatly reduce the number of parameters and computational effort. It is worth noting that when choosing the pruning method, we abandoned the method with a higher compression rate and chose row pruning which is more convenient for hardware computation, which paves the way for the following hardware design.

## 4 Hardware Design

There are many hardware designs for LSTM, but most of them did not consider the characteristics of the hardware structure afterward in the process of software optimization. Therefore, the final design of the hardware structure was often limited by the characteristics of the model and data. In this paper, we take into account the design characteristics of the hardware structure in the process of model compression and choose a compression method that is more compatible with our hardware design to ensure hardware friendliness to the maximum extent.



**Fig. 2.** Overall architecture diagram

#### 4.1 Overall Structure

The overall architecture of the hardware is shown in Fig. 2. The entire hardware architecture consists of the controller, the input and output buffers, BRAM, and LSTM computation blocks. The ARM core controls the overall flow of the model by transmitting instructions to the hardware's controller. In addition, the parameters and input data required for the operation are pre-processed by model compression and stored in BRAM. The controller accepts instructions from the ARM core and then coordinates and calls other areas according to the instructions. The input and output buffer temporarily stores the data in the BRAM to reduce the data transfer latency of the LSTM computation block. The LSTM computational block implements the computational logic of multi-layer LSTM. Each neuron in this unit corresponds to and implements the computational flow of a single neuron in the LSTM algorithm, while the LSTM neurons are densely connected by input and output ports to implement the data path between neurons.

#### 4.2 Compute Optimization

The main computations of LSTM cells are four matrix multiplications, activation functions, dot-product, and addition. Our optimization scheme focuses on matrix multiplication, which accounts for a large part of the overall compute. We block the matrix by rows and perform the input in parallel. In matrix computation, the internal loop is unrolled when the elements of multiple rows are simultaneously assigned to different PE. Since the pruning by row in model compression allows the same number of elements in each row of the weight matrix, it ensures that the data assigned to different PEs in different rows have the same computation time. After computing finishes, PE can proceed directly to the next cycle without

waiting for other PEs. Given the limited resources on the hardware, the number of parallel PEs can be fine-tuned according to the application scenario. If PE resources are sufficient, we reduce the time complexity to  $O(N)$  in an ideal case.

The computation process of an LSTM cell is shown in Fig. 3. On the left is the processed weight matrix. We divide it into rows and assign them to different PEs. And on the top is the input vector  $x$  and the previous neuron input  $h_{t-1}$ . After the input parameters  $(h_{t-1}, x_t, c_{t-1})$  enter LSTM cell,  $h_{t-1}, x_t$  first carry out matrix multiplication and addition operations with the four weight matrices  $(W_f, W_i, W_c, W_o)$  and then obtain the intermediate results  $(i_t, f_t, g_t, o_t)$  after  $\sigma$  and  $\tanh$  activation functions. The compute process of LSTM includes cell state  $c_t$  and hidden state  $h_t$ . The cell state  $c_t$  is obtained from the dot product of  $i_t$  and  $g_t$  plus the dot product of  $f_t$  and another input parameter  $c_{t-1}$ , while the hidden state  $h_t$  is the result of  $c_t$  after  $\tanh$  and then dot product with  $o_t$ . The computation of input parameters and weight matrix is the most time-consuming portion of the entire procedure. Since the four matrix calculations are independent, this paper replicates the input parameters so that the four matrix operations are no longer limited by the transmission delay of the input parameters and are computed simultaneously, which reduces the overall computation time.

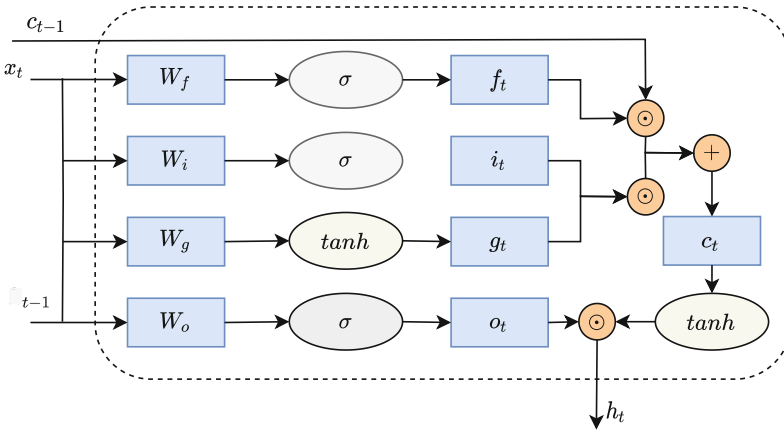


Fig. 3. Computational process of a single LSTM neuron.

### 4.3 Data Flow Optimization

We analyze the data flow between each neuron in the layer dimension and time dimension of the LSTM. The LSTM of the same layer is executed sequentially in 2 dimensions on the CPU, which leads to low parallelism of the model. In the hardware implementation, the strong timing of the LSTM results in a parallel strategy that can only be carried out in two separate dimensions including, between different layers of the same sequence, such as Layer1.1, Layer2.1



and Layer3\_1; And between different times of the same layer, such as Layer1\_1, Layer1\_2 and Layer1\_3.

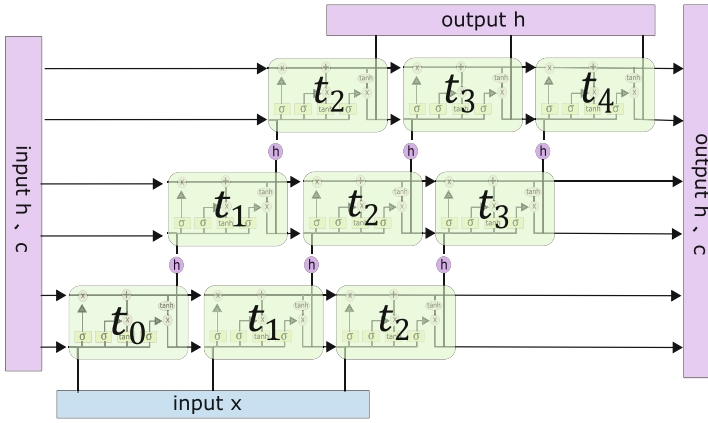


Fig. 4. Multi-layer LSTM data stream (take 3 layers as an example).

The distinction between hardware and software implementations inspires us. Hardware LSTMs often choose to implement each cell computation in turn and spread it out to match the rich hardware resources, while software calls the same cell module in turn. This difference in implementation brings a new perspective on the data flow. As shown in Fig. 4, when the first time step of the first layer LSTM is computed at  $t_0$  moment, the output  $h_0$  at the moment  $t_0$  is passed to the second time step of the first layer and the first time step of the second layer LSTM after completion of the computation. After that, at  $t_1$  moment, the first time step of the second layer LSTM and the second time step of the first layer LSTM can compute simultaneously, and so on. Thus some cells of LSTM can compute in parallel to achieve acceleration, which is similar to the acceleration idea of the pulsating array [29]. After optimizing the computational order, we can observe that the overall model is more parallelized. As a result, we can adjust the timing of the cells that satisfy the computational prerequisites at the same moment in the hardware implementation. Figure 5(a) shows the timing diagram before the adjustment, where the whole computation process took nine clock cycles, while in Fig. 5(b), the computation process took only five clock cycles after the adjustment. For this stepwise computation order, Layer1\_3, Layer2\_2, and Layer3\_1, which are in different layers, start running at the same time because they satisfy the computation prerequisites at the same time, thus satisfying the two parallel strategies at the same time.

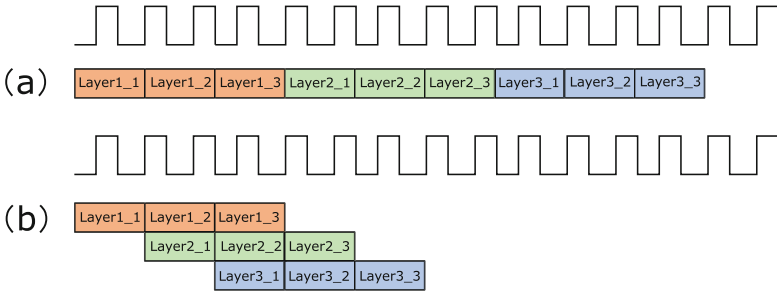


Fig. 5. Pipeline structure of multi-layer LSTM.

## 5 Experimental Results and Analysis

We implemented the LSTM neural network on the MNIST [30] dataset using the pytorch framework. The LSTM neural network was trained on the MNIST dataset, which consists of four main components: the training set, the test set, the validation set images and the label information. There are 55,000 images in the training set, each with a size of 784 (28\*28); 55,000 labels in the training set, each label being a one-dimensional array of length 10; 10,000 images in the test set and 5,000 images in the validation set.

The experimental results showed that the accuracy before pruning reached 95.75%. After pruning the weight matrix, the compression rate reached 21.99×, but as we were pruning uniformly by row, this inevitably led to the loss of valid parameters and the accuracy dropped to 68%, such a reduction in accuracy is unacceptable. We therefore retrained after the pruning was completed, and the final accuracy was 96%. Finally, we had completed the effective compression of our model.

We implemented the LSTM computation process on Vivado High-Level Synthesis V2020.2 and the results are shown in Table 1. The device we chose was the XC7Z020, and in order to make the best use of hardware resources, we explored the relationship between the size of the weight matrix and the consumption of hardware resources and its percentage of the total amount of each resource. It is clear from Table 1 that when the size of the weight matrix is 32 × 18, the percentage of BRAM\_18 K, DSP and FF usage is small, but the LUT usage reaches 27132, accounting for 51% of the total resources.

As the size of the weight matrix increases, all the hardware resources increase with it, except for FF, whose percentage did not change much. Among them, BRAM\_18 K increases the most, directly from the initial 34% to 103%. As the scale increases to 64 × 256, the resources required for BRAM\_18 K exceed the total resources available from the hardware. When the model size is 50 × 200, the proportion of each resource is more reasonable.

**Table 1.** Hardware resource usage for different model sizes

Matrix size	BRAM_18K	DSP	FF	LUT
$32 \times 18$	48 (34%)	26 (12%)	12768 (12%)	27132 (51%)
$50 \times 200$	66 (47%)	99 (45%)	28728 (27%)	30324 (57%)
$64 \times 256$	144 (103%)	130 (59%)	24472 (23%)	37242 (70%)

We compared the hardware resource consumption before and after optimization and computed the percentage of total resources on the hardware accounted for by LUT, FF, DSP, and BRAM.18 K. Before optimization, LUT and BRAM.18 K account for a large proportion of the hardware resource consumption, accounting for 52% and 34% of the total number of resources, respectively. Due to the use of various hardware optimization schemes, the latency is reduced while the consumption of various resources is significantly increased. Since LSTM compute contains many matrix multiplications, our optimized DSP has the largest increase in consumption with 40%, accounting for 70% of the total DSP resources.

## 6 Conclusion

In this paper, we proposed the method of optimizing neural networks based on the idea of hardware-software collaboration for multilayer LSTM. The selection of the pruning scheme and the corresponding hardware design in this paper fully reflected the idea of hardware-software collaboration. After analyzing the data flow and data dependency of the LSTM, we redesigned the computational order of the multilayer LSTM to parallelize the serial computation of some neurons. The final experiment proved that our solution had 10× the throughput of CPU and improved in terms of throughput, latency, and resource utilization compared to other hardware accelerators.

## References

1. Qiu, M., Xue, C., et al.: Energy minimization with soft real-time and DVS for uniprocessor and multiprocessor embedded systems. In: IEEE DATE, pp. 1–6 (2007)
2. Qiu, M., Xue, C., et al.: Efficient algorithm of energy minimization for heterogeneous wireless sensor network. In: IEEE EUC, pp. 25–34 (2006)
3. Qiu, M., Liu, J., et al.: A novel energy-aware fault tolerance mechanism for wireless sensor networks. In: IEEE/ACM Conference on GCC (2011)
4. Wu, G., Zhang, H., et al.: A decentralized approach for mining event correlations in distributed system monitoring. JPDC **73**(3), 330–340 (2013)
5. Lu, Z., et al.: IoTDeM: an IoT big data-oriented MapReduce performance prediction extended model in multiple edge clouds. JPDC **118**, 316–327 (2018)

6. Qiu, L., Gai, K., Qiu, M.: Optimal big data sharing approach for tele-health in cloud computing. In: IEEE SmartCloud, pp. 184–189 (2016)
7. Zaremba, W., Sutskever, I., Vinyals, O.: Recurrent neural network regularization (2014). arXiv preprint, [arXiv:1409.2329](https://arxiv.org/abs/1409.2329)
8. Hochreiter, S., Schmidhuber, J.: Long short-term memory. *Neural Comput.* **9**(8), 1735–1780 (1997)
9. Graves, A., Schmidhuber, J.: Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Netw.* **18**, 602–610 (2005)
10. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to forget: Continual prediction with LSTM. *Neural Comput.* **12**(10), 2451–2471 (2000)
11. Donahue, J., Anne Hendricks, L., et al.: Long-term recurrent convolutional networks for visual recognition and description. In: IEEE CVPR, pp. 2625–2634 (2015)
12. Bank-Tavakoli, E., Ghasemzadeh, S.A., et al.: POLAR: a pipelined/overlapped FPGA-based LSTM accelerator. *IEEE TVLSI* **28**(3), 838–842 (2020)
13. Liao, Y., Li, H., Wang, Z.: FPGA based real-time processing architecture for recurrent neural network. In: Khafa, F., Patnaik, S., Zomaya, A.Y. (eds.) IISA 2017. AISC, vol. 686, pp. 705–709. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-69096-4\\_99](https://doi.org/10.1007/978-3-319-69096-4_99)
14. Chang, X.M., Culurciello, E.: Hardware accelerators for recurrent neural networks on fpga. In: IEEE Conference on ISCAS, pp. 1–4 (2017)
15. Li, S., Wu, C., et al.: Fpga acceleration of recurrent neural network based language model. In: IEEE Symposium on Field-Programmable Custom Computing Machine, pp. 111–118 (2015)
16. Li, Y., et al.: Intelligent fault diagnosis by fusing domain adversarial training and maximum mean discrepancy via ensemble learning. *IEEE TII* **17**(4), 2833–2841 (2020)
17. Qiu, H., Zheng, Q., et al.: Deep residual learning-based enhanced jpeg compression in the internet of things. *IEEE TII* **17**(3), 2124–2133 (2020)
18. Gai, K., et al.: Efficiency-aware workload optimizations of heterogeneous cloud computing for capacity planning in financial industry. In: IEEE CSCloud (2015)
19. Qiu, M., et al.: Data transfer minimization for financial derivative pricing using monte carlo simulation with GPU in 5G. *JCS* **29**(16), 2364–2374 (2016)
20. Qiu, H., et al.: Topological graph convolutional network-based urban traffic flow and density prediction. *IEEE TITS* **22**, 4560–4569 (2020)
21. Qiu, H., et al.: Secure health data sharing for medical cyber-physical systems for the healthcare 4.0. *IEEE JBHI* **24**, 2499–2505 (2020)
22. Qiu, M., Zhang, L., et al.: Security-aware optimization for ubiquitous computing systems with seat graph approach. *JCSS* **79**(5), 518–529 (2013)
23. Qiu, M., Li, H., Sha, E.: Heterogeneous real-time embedded software optimization considering hardware platform. In: ACM SAC, pp. 1637–1641 (2009)
24. Ferreira, J.C., Fonseca, J.: An FPGA implementation of a long short-term memory neural network. In: IEEE Conference on ReConFigurable, pp. 1–8 (2016)
25. Guan, Y., Yuan, Z., Sun, G., Cong, J.: Fpga-based accelerator for long short-term memory recurrent neural networks (2017)
26. Ledwon, M., Cockburn, B.F., Han, J.: High-throughput FPGA-based hardware accelerators for deflate compression and decompression using high-level synthesis. *IEEE Access* **8**, 62207–62217 (2020)
27. Han, S., Mao, H., Dally, W.J.: Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding (2015). arXiv preprint, [arXiv:1510.00149](https://arxiv.org/abs/1510.00149)

28. Han, S., Kang, J., et al.: ESE: Efficient speech recognition engine with sparse LSTM on FPGA. In: ACM/SIGDA FPGA, pp. 75–84 (2017)
29. Jouppi, N.P., Young, C., et al.: In-datacenter performance analysis of a tensor processing unit. In: 44th IEEE ISCA, pp. 1–12 (2017)
30. Yadav, C., Bottou, L.: Cold case: The lost mnist digits (2019)