Implementing Neural Machine Translation with Bi-Directional GRU and Attention Mechanism on FPGAs Using HLS

Qin Li[1*], Xiaofan Zhang[1*], JinJun Xiong[2], Wen-mei Hwu[1], Deming Chen[1]
[1]University of Illinois at Urbana-Champaign, [2]T.J. Watson Research Center, IBM
{qinli2, xiaofan3, w-hwu, dchen}@illinois.edu, jinjun@us.ibm.com

ABSTRACT

Neural machine translation (NMT) is a popular topic in Natural Language Processing which uses deep neural networks (DNNs) for translation from source to targeted languages. With the emerging technologies, such as bidirectional Gated Recurrent Units (GRU), attention mechanisms, and beam-search algorithms, NMT can deliver improved translation quality compared to the conventional statistics-based methods, especially for translating long sentences. However, higher translation quality means more complicated models, higher computation/memory demands, and longer translation time, which causes difficulties for practical use. In this paper, we propose a design methodology for implementing the inference of a real-life NMT (with the problem size = 172 GFLOP) on FPGA for improved run time latency and energy efficiency. We use High-Level Synthesis (HLS) to build high-performance parameterized IPs for handling the most basic operations (multiply-accumulations) and construct these IPs to accelerate the matrix-vector multiplication (MVM) kernels, which are frequently used in NMT. Also, we perform a design space exploration by considering both computation resources and memory access bandwidth when utilizing the hardware parallelism in the model and generate the best parameter configurations of the proposed IPs. Accordingly, we propose a novel hybrid parallel structure for accelerating the NMT with affordable resource overhead for the targeted FPGA. Our design is demonstrated on a Xilinx VCU118 with overall performance at 7.16 GFLOPS.

1 INTRODUCTION

Machine translation is one of the most challenging applications for natural language processing. During recent years, we have seen rapid development of technologies which can be applied to machine translation and among them, NMT, a neural network based solution has become one of the best solutions which outperforms the conventional statistical machine translation [1–4]. The NMT takes advantages of its inherent ability of exploiting and understanding the whole input sentences before generating the output sentences so that it can deliver more meaningful and more fluent translations.

*These authors made equal contributions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASP-DAC 2019, Jan. 2019, Tokyo, Japan

© 2019 Association for Computing Machinery.

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

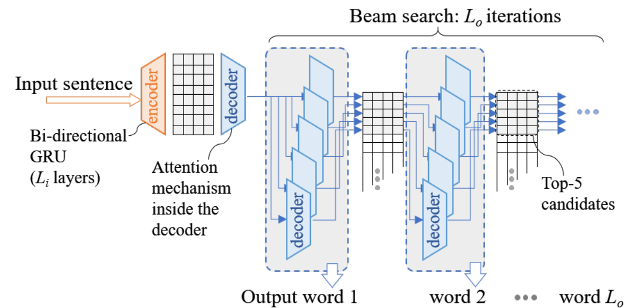

Figure 1: Structure of a real-life NMT with encoder-decoder structure, attention mechanism, and beam search.

In most practical cases explored so far, NMT has been implemented using the encoder-decoder algorithm [3, 4]. To overcome the vanishing gradient problem, Long Short Term Memory (LSTM) and GRU are proposed to better reserve the long-term memory so that NMT can understand the information coming from long sentence inputs [3, 4]. However, the encoder-decoder based NMT models pose a great challenge to encode any sentences into fixed-length vectors, which means that information needs to be compressed when input becomes longer. As a result, the quality of the output produced deteriorates significantly as the length of the input sentence increases. Hence, an attention mechanism is needed to allow NMT to pay more attention to the more relevant portions, i.e., certain word or phrase, instead of focusing on the whole input sentence [2]. Also, beam search is often used for improving the output candidate selection.

Although NMT is a powerful tool, it is very complicated and difficult to implement in hardware. Figure 1 presents a high-level view of the real-life NMT model whose hardware implementation is the focus of this paper. It combines the most popular encoder-decoder structure with an advanced attention mechanism in each decoder and a beam search algorithm for picking the top-N candidates ($N = 5$ in Figure 1) every iteration for better adaption of the long sentence inputs. The length of input and output sentences are donated as L_i and L_o, so that this model have L_i layers of bidirectional GRU and L_o iterations in beam search. In real-life applications, L_i and L_o can even reach 50, so that we set both L_i and L_o to 50 in our design.

In this paper, we aim to exploit the capability of FPGAs for delivering high energy efficiency while parallelizing tasks for such compute intensive real-life NMT applications. We leverage HLS to explore the design space to propose an energy-efficient, high-performance FPGA-based NMT design with a bidirectional GRU, attentional mechanism, and beam search algorithm. To summarized, the main contributions of this paper are:

(1) **The first FPGA-based design of a real-life NMT model**. We implement a real-life NMT model with all key features including bidirectional GRU, attention mechanism, and beam search algorithm.

(2) **Comprehensive design space explorations using HLS**. We explore the design space for implementing NMT under FPGA's computation and memory constraints. Different parallelism degrees from various levels (e.g., IPs, kernels, modules) are evaluated to finalize the best configurations.

(3) **Highly optimized HLS IPs**. We propose major types of HLS IPs as basic building blocks for implementing NMT. With the parameterized IPs, functions can be configured according to the resource exploration results for the best performance.

(4) **A hybrid parallel structure for running NMT with affordable resource overhead and high performance**. We combine the recurrent (folded) structure with parallelized decoders for a better tradeoff between resource utilization and performance.

The rest of this paper is organized as follows. In Section 2 and 3, we introduce the related work and the targeted NMT model respectively. The proposed FPGA design and optimization schemes are described in Section 4 and 5. Experimental results are provided in Section 6. In Section 7, we conclude this paper.

2 RELATED WORK

FPGA has demonstrated its effectiveness by successfully implementing complex applications such as H.264 video decoder [5] and object-detection system [6]. As DNN-applications become one of the most complicated workloads available today, a significant amount of work have been done for mapping DNNs to FPGAs which cover both Convolutional Neural Networks (CNNs) [7–9], Recurrent Neural Network (RNNs) [10–12], and even the hybrid structure containing both CNNs and RNNs [13]. An accelerator for VGG network is investigated and mapped onto FPGA in [7] using OpenCL design flow. In [8], a Winograd-based template is used to reduce the number of multiplications and boost the CNN inference performance on FPGA. A automation tool for building FPGA-based CNN accelerator is developed in [9] which bridges the gap between DNN design in machine learning frameworks and FPGA board-level implementation. Previous literature also focuses on implementing RNNs on FPGAs. A LSTM model compression scheme is proposed in [10] for exploring the sparsity of neural network and building FPGA-based accelerator. In [11], a RNN with LSTM layers is implemented on FPGA with 7.26 GFLOPS peak reported performance. However, the authors did not provide the detailed conditions for reaching such performance while the end-to-end throughput is only 0.007 GFLOPS (problem set size: 2.76 mega-operations divide by overall latency: 0.39 second). A LSTM with fixed-point data type is implemented in [12] because FPGAs perform favorably with fixed-point DSP units but not as well when using floating-point DSP units. In addition, an accelerator and a corresponding resource allocation scheme for Long-term Recurrent Convolutional Network are present in [13] which covers both CNN and RNN implementations on FPGA.

Although FPGAs have shown great potentials on DNN implementation, there is still no literature focusing on building an end-to-end NMT model on FPGA for running natural language translation. Based on a deep RNN-like structure (e.g., LSTM and GRU with around 50 layers), the real-life NMT employs more hidden neurons (≥1024 neurons every layer) and intricate structures (such as attention mechanism and beam search algorithm) for better understand the input sentences. These features, however, make NMT difficult to design and implement in hardware.

The novelty in this project lies in the FPGA implementation of the latest NMT model featuring bidirectional GRU, attention mechanism, and beam search. We propose the highly optimized HLS IPs and use them to build the MVM kernel for dealing with the most computational intensive parts. By using HLS, we also perform design space exploration to evaluate the configurations for building MVM kernel. In addition, we propose a FPGA resource distribution scheme for workload balancing across layers so that we tradeoff the computation demands and the limited resources available. To the best of our knowledge, this is the first such an implementation on FPGAs.

3 NMT

The general idea of NMT is to first pre-process the input sentence with length L_i into L_i word-embedding vectors, and then go through encode-decoder structure and finally output the translated sentence. The characteristic functions of a real-life NMT can be found in the encoder (with bidirectional GRUs) and the decoder (with attention and beam search algorithm).

3.1 Bidirectional GRUs

$$z^{(t)} = \sigma(W^{(z)}x^{(t)} + U^{(z)}h^{(t-1)}) \tag{1}$$

$$r^{(t)} = \sigma(W^{(r)}x^{(t)} + U^{(r)}h^{(t-1)}) \tag{2}$$

$$\tilde{h}^{(t)} = \tanh(U(r^{(t)} \circ h^{(t-1)}) + Wx^{(t)}) \tag{3}$$

$$h^{(t)} = (1 - z^{(t)})h^{(t-1)} + z^{(t)}\tilde{h}^{(t)} \tag{4}$$

The RNNs we used for encoder are GRUs with 1024 neurons. There are two gates in the GRU unit: the update gate and the reset gate whose functions illustrated by Eq. (1) and Eq. (2), respectively. The update gate controls the amount of memory need to be updated while the reset gate determines the portion of memory need to be discarded. Therefore, after running Eq. (3) and Eq. (4), the output of current state is generated, and it will be used as the memory for next state. Noticed the $\circ$ is an element-wise multiplication.

In general, RNNs learn information from previous states. However, this learning behavior is not good enough for handling natural language translation since the meaning of a word depends on not only the text shown before but also after it. Therefore, NMT model uses bidirectional RNN (BRNN) to link both parts preceding and following a word. According to [14], bidirectional RNNs can deliver higher accuracy comparing to single layer of RNNs.

BRNN contains two layers of RNNs as forward RNN and backward RNN (Figure 2). The forward RNN reads inputs in an ordered way while the backward RNN reads reversely. BRNN generates forward hidden states and backward hidden states annotating the same input word with $h = \{h_f, h_b\}$ (each vector h has size of $1024 \times 2 = 2048$). After processing all input words, BRNN outputs a vector map with size of $2048 \times L_i$, annotating the whole input sentence.

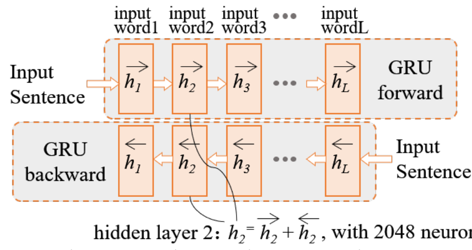

Figure 2: Bidirectional GRU layers in the targeted NMT encoder

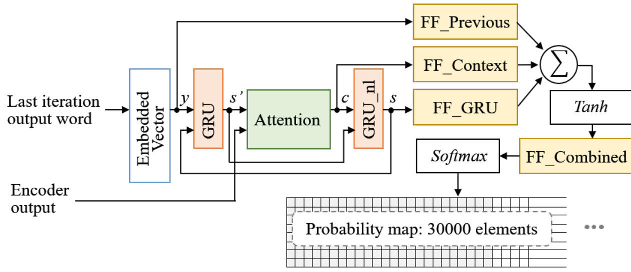

Figure 3: Decoder structure

3.2 Decoder Structure

Opposite to the encoder, the decoder converts the vector into a meaningful sentence in targeted language using RNNs. Figure 3 shows a detailed structure of the decoding process. The embedded vector representing the last generated word is passed into the first GRU layer and then sent to the attention module. The output of attention module (c) is put into another GRU layer (GRU_nl) to generate hidden state s. After that, the context c, the embedded vector y, and the second GRU output s are passed to three feed-forward (FF) layers, respectively. The vector generated by those FF layers are combined together and put into another FF layer (FF_Combined). After performing softmax of the output, a probability map is generated with the same size as the length of the target dictionary (30000 in our case) .

3.3 Attention Mechanism

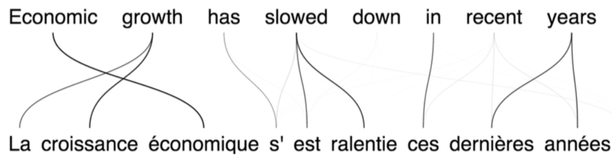

Figure 4: Example of a pair of sentences showing how the words in source sentence are mapping to the target sentence. The darker the line is, the more related the pair of words linked by the line [15].

The transformation from the source language to the target language is not purely one-to-one mapping. Multiple words from source language may be related to a single word in the target language and vice versa. Hence, we need attention mechanism

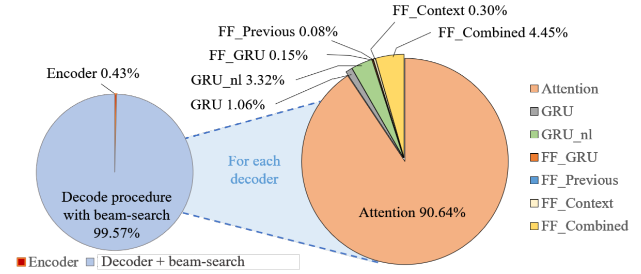

Figure 5: Operation distribution in the targeted NMT (left) and computational demand breakdown in a single decoder (right)

to determine the importance of each word in source language while generating a target word. Detailed algorithm can be found in Eq. (5) through (7).

$$e_{ij} = v^\top \tanh(U_a s'_j + W_a h_i) \tag{5}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_k^L \exp(e_{kj})} \tag{6}$$

$$c_j = \sum_i \alpha_{ij} h_i \tag{7}$$

In Eq. (5), s'_j represents the hidden state generated by the first GRU in the decoder, while h represents each hidden state of the encoder. The subscript i and j indicate the number of iteration of the encoder and the decoder, respectively. V_a, U_a, and W_a are constants in these equations while e_{ij} represents the energy state. In Eq. (6), all energy states generated by iteration j are normalized using softmax, transferring to α_{ij} to report the importance of each input word during translation. Finally, the attention information c is generated in Eq. (7).

3.4 Beam Search

The goal of NMT is to find the most suitable output sentence. However, previous works as in [3] have demonstrated that only choosing one word with the highest probability for each iteration may not have satisfactory results. Instead, the recent NMT uses beam search for selecting more candidate words in each iteration and improving the accuracy. We choose beam width $K = 5$ in our implementation. For each iteration, the top five candidates are chosen as potential answers. Whenever the generated word is *eos* (end of sentence) is seen, the activated decoder is no longer propagated. The whole process terminates when all of the five paths reach *eos*, thereby selecting the sentence with the highest score.

4 DESIGN METHODOLOGIES

In this section, we summarize the design methodologies during NMT implementation regarding the complicated network interconnection, computational demand, and the utilization of HLS design flow.

4.1 Profiling Results

According to the algorithm mentioned in section 3, GRU layers, attention mechanism, and feed-forward layers are heavily used

in the model. We first encapsulate these network features into separated functions and use those functions as basic building block to construct our design using HLS.

The NMT model we targeted is a real-life model, which employs complicated and computation-intensive network structures. It places very high demand on computation with 172 GFLOP in total for translating a 50-word sentence. Inside the encoder, there are bidirectional GRUs (with 2048 neurons in each layer and L_i layers in total) as shown in Figure 2. Regarding the decoder-side, attention mechanism is used for each decoder and because of the beam search algorithm, the encoded features need to go through $5L_o + 1$ decoders (Figure 1) for generating final results. The whole NMT flow needs to consume a great amount of hardware resource and it is challenging to be mapped onto FPGA.

To understand the computational complexity distribution of the NMT model, we start profiling and collect the number of operation in each layer. Results are shown in Figure 5, the amount of operation in encoder is negligible (0.42%) compared to the decoder, which consumes 99.67% of the total complexity. Therefore, our optimization mainly focuses on the decoder side. In our calculation, we assume the length of input and output sentences are the same as $L_o = L_i = 50$.

4.2 NMT overall design

The structure of our decoder design is shown in Figure 6(a). All of the intermediate results are stored in on-chip buffers, while weights and biases are stored off-chip because of limited size of the on-chip BRAM. Therefore, during each calculation, portions of weights need to be loaded from DDR memory. We developed some techniques to exhaustively reuse data as well as resources on the board.

To fully utilize the on-chip memory in the FPGA because of its fast data access, we want to try our best to reuse the idle portion. Once the data is propagating through each layer, the intermediate result can be stored in the same set of buffers. For instance, the number of neurons in each GRU layer of the NMT model is the same. Therefore, one set of buffers can be shared for multiple GRU layers. The buffer sharing technique allows intermediate results of such a large NMT model to be fully stored on-chip.

Theoretically, when the data is propagating through a specific layer, computation resources for other layers will be idle if they are not shared. In order to avoid enormous idle IPs, we want to share them for multiple layers. In this way, the DSP usage is reduced. However, enormous MUXes need to be instantiated for this purpose. We have to control the number of IPs instantiated for the whole network to balance between DSP usage and LUT usage.

4.3 Highly Optimized IPs

Most of computation is induced by MVM(Matrix Vector multiplication) processes in our implementation. For the acceleration purpose, we developed an optimized IP for MVM kernels with fixed size of input/output buffers and use the IP globally for MVM kernels.

The IP consists of several compute engines(CE). These engines are optimized implementations of MAC units with registers storing the intermediate data. The multiplied results are added using an adder tree, generating a complexity of $O(\log n)$. All of the input arrays are partitioned so that all of the CEs can be launched in parallel and all of the multiplication processes run simultaneously.

Table 1: Effect of CE size

CE size	latency	DSP%
32	21	2
64	25	4
128	29	9

Large matrix vector multiplication tasks can be assigned to several small MVM processes. As shown in Figure 7, in a MVM process, the matrix M and vector V are cut into smaller portions to fit the input size of the IP first, and then one set of those portions will be stored in buffers during each calculation. The size of buffers are designed to fit the size of the IP inputs and outputs. After each calculation, the output of the kernel will be added back to correct output dimension.

Two kinds of MVM structures are designed in order to align with the beam search algorithm. One deals with single MVM process and the other one can handle multiple MVM processes (named MVM_five) that share the same weight. At the first iteration of the decoder as well as all the MVM processes occurred in the encoder, only one MVM process will be performed each time.When the decoder reaches the second stage, multiple decoding processes need to be performed in the later iterations because of the beam search algorithm.

4.4 Hybrid Parallel Decoders

In order to increase the data reuse and eliminate unnecessary data loading, we developed a hybrid parallel structure. Multiple MVMs are processed portion by portion, as shown in Figure 6(b).

Since we choose five as the width of the beam search, at most five decoding processes will be activated. Since all of the decoders share the same set of weights (stored off-chip), we only have to load one portion of weight to perform five calculations for decoders if the decoders run in parallel. The design can increase the data reuse compared to series design.

When a portion of weight is loaded off-chip, the computation task is to calculation five portions of MVM processes in five decoders respectively. Since intermediate results can be stored in registers instantiated in CEs, as long as the computation begins, we are able to load next set of data into the IP buffers. Therefore, we have to find a suitable IP size in order to achieve comparable loading time and computation time so that the loading time can hide the computation time. The IP size is related to CE size as well as the number of CEs instantiated. The number of CEs is not limited to five to make the five decoders run exactly in parallel. We can firstly load all five sets of data, and then perform the calculation in series depending on how many CEs are instantiated.

5 DESIGN SPACE EXPLORATION

NMT is a large network containing various layers and a huge number of parameters. How to properly allocate the limited resources in FPGA to achieve a complete network or even a good performance becomes really important. Most of the calculation concerns about the MVM process. Therefore, we conduct an experiment to explore how the MVM kernel should be designed to achieve the best performance.

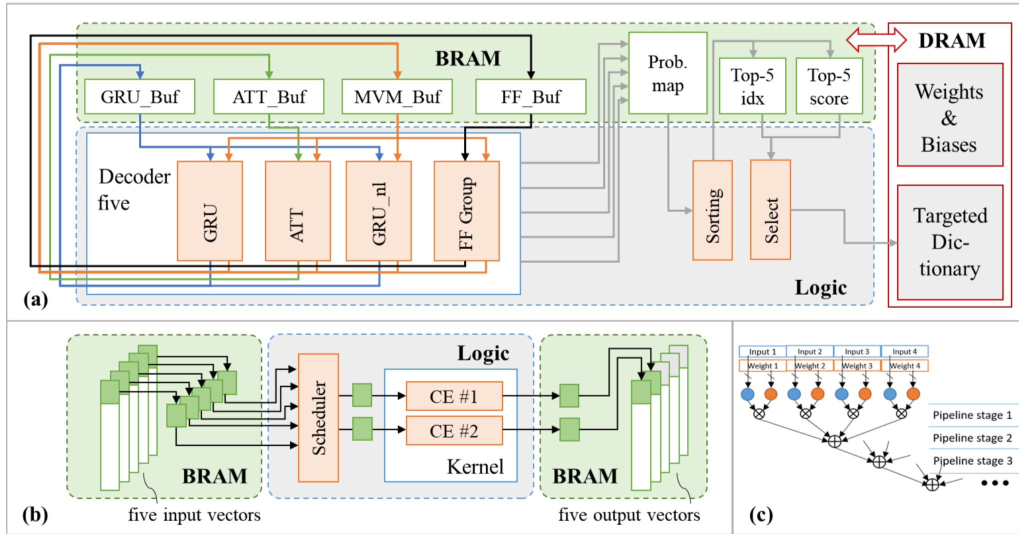

Figure 6: (a) Architecture level structure of decoder with beam search algorithm applied; (b) Hybrid parallel MVM structure; (c) CE with Fine-grained pipeline stages

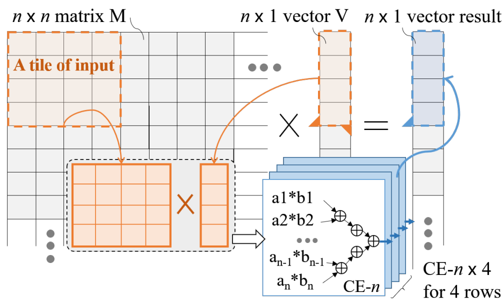

Figure 7: Example for MVM process with four CEs instantiated

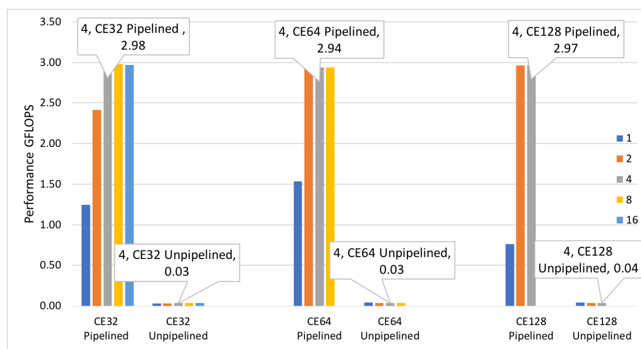

Figure 8: Performance of different kernel design in single 512 MVM process

Three procedures are included in each iteration of this MVM process. First, the weight needs to be loaded into buffers, then the computation IP will do the calculation. After that, the results produced by the IP will be added to the output. The computation IP is made of single or multiple CEs. Considering limited resources in FPGA, we firstly analyze the resource for different sizes of CEs. As shown in table 1, the DSP usage is proportional to the CE size.

We instantiated different computation IPs and recorded the performance when those IPs calculated vector matrix multiplication with vector with size of 512 by 1 and matrix with size of 512 by 512, as shown in Figure 8. In the Figure, different colors of bars represent different numbers of sets of buffers instantiated. Both pipelined version and unpipelined version are included in the graph.

Pipelined vs unpipelined: The CE contains registers that can save intermediate results. Once the loading process is done, the calculation can be launched. We ignore the storing time because the number of stored data in each process is negligible. After the first stage, the intermediate results are stored in the registers. At that time, the next set of data can start to be loaded to the buffers. Therefore, with pipelining, the latency can be largely reduced. As shown in Figure 8, the performance for pipelined version is much higher than that of the unpipelined version.

Furthermore, since the CE we developed has internal registers to store intermediate results, one single CE can be pipelined so that it can handle multiple sets of data. From the analysis using HLS tool, we find out that one pipelined CE can achieve similar performance compared to multiple CEs run in parallel if they have the same sets of buffers. However, when the CE is not pipelined, it can not simultaneously handle multiple sets of data. Therefore, for the pipelined version, we only instantiated single CE to handle various sets of data. For the unpipelined version, the number of CEs and the number of buffers are the same.

Match between loading and computation: As shown in the Figure 8, the performance will remain nearly the same if the number of buffer exceeds certain limit for the same size of CE. The process is limited by time for loading off-chip data. In order to achieve the best performance, the time for data loading and calculating should be similar, large portion of them can be overlapped under pipelining. In that way, we can achieve the good performance as well as efficient utilization.

Table 2: Comparison with previous work

Reference	[11]	our work
Targeted Model	3 LSTM Layers	A real-life NMT
Total Size	2.76MOP	170GOP
FPGA type	Virtex-7	VCU118
Precision	Float32	Float32
DSP Usage	1176	5969
Frequency	150MHz	100MHz
Peak performance	7.26 GFLOPS	14.8 GFLOPS (2x)
End-to-end performance	0.007 GFLOPS	7.16 GFLOPS (100x)

For MVM_five, since the weights are shared, comparing to the single MVM version, loading speed can be considered five times faster than that of single MVM version. Therefore, with matched MVM_five kernel, we can achieve at most five times speed up. We achieved 14.8 GFLOPS in our hybrid parallel MVM_five process.

Choosing IP structure: Firstly, since the network is pretty huge, with limited data reuse, the resource usage limits both the number of IP instantiated and the size of the IP. Also, extra DSP is needed for non-linear calculations such as exponential and hyper tangent. We want to choose a set of CEs that can achieve both high performance and efficient resource usage.

6 EXPERIMENTAL RESULTS

In this section, we demonstrate our NMT design on a Xilinx Ultrascale+ VCU118 development board with a XCVU9P FPGA. We first introduce the prepare work of this project and then we present the performance results and compare our design to the previous literature.

6.1 Prepare work

To implement the overall NMT model, we first converted the python source code into synthesizable C code without any optimization in HLS. After obtaining a baseline version, we applied our design optimization to speed up the performance.

We used 32-bit floating point, same datatype as the original model, to complete the design. Even though FPGA cannot efficiently deal with floating point calculation, using the same data format can avoid accuracy loss.

6.2 Results and comparison

After constructing the whole model, we compared our work with previous RNN related hardware implementation. As shown in Table 2, our work has more balanced performance with our optimized pipelined structure compared to the other work since our peak performance and the end-to-end performance (average performance of the entire flow) are relatively close to each other. Also, with kernel reuse, the DSP usage does not increase a lot even though our model is much larger.

The original NMT model is constructed by Theano, a platform designed for deep learning. We tested the performance for our FPGA version by using an input sentence with length 50. As shown in Table 3, the latency is compared with performances of two CPUs using the same sentence. For CPUs, the weights are firstly loaded into cache and then the data is computed. Since FPGA has limited memory to store the weights (total size of 300+MB), loading and calculation are done portion by portion simultaneously.

Table 3: Performance Comparison with Other Devices

Hardware Type	loading	computation	total
Intel(R) Xeon(R) CPU E5-2603	64.9s	18.7s	87.7s
Intel(R) Xeon(R) i5 CPU 650	66.0s	8.95s	74.95s
XCVU9P FPGA	(loading + computation) 24.0s		

7 CONCLUSION

In our work, we built the first FPGA-based implementation of the Neural Machine Translation model. Unlike previous works which play with small-scale RNN or LSTM models, we implement a real-life NMT model with all latest features including bidirectional GRU, attention mechanism, and beam search algorithm. We present a comprehensive design space exploration to better deliver high performance NMT design under FPGA resource constraints. By taking advantage of HLS, we can alleviate the design and optimization efforts.

For the future work, we can focus on converting the model to half-floating point or even fixed point with less bit width than single precision for further acceleration purpose, because they allow larger number of weights transfer per unit time under limited memory bandwidth. Furthermore, we can do quantization and linear approximations. By retraining the model, we could potentially gain high performance and retrieve accuracy back after applying those optimization techniques.

ACKNOWLEDGMENTS

This work was partly supported by the IBM-Illinois Center for Cognitive Computing System Research (C^3SR) – a research collaboration as part of IBM AI Horizons Network.

REFERENCES

[1] Nal Kalchbrenner and Phil Blunsom. Recurrent continuous translation models. In *EMNLP*, 2013.
[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2015.
[3] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.
[4] Kyunghyun Cho et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv:1406.1078*, 2014.
[5] Xinheng Liu et al. High level synthesis of complex applications: An h. 264 video decoder. In *FPGA*, 2016.
[6] Su Liu et al. Real-time object tracking system on fpgas. In *Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, 2011.
[7] Jialiang Zhang and Jing Li. Improving the performance of opencl-based FPGA accelerator for convolutional neural network. In *FPGA*, 2017.
[8] Junzhong Shen et al. Towards a uniform template-based architecture for accelerating 2d and 3d cnns on FPGA. In *FPGA*, 2018.
[9] Xiaofan Zhang et al. Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas. In *ICCAD*, 2018.
[10] Song Han et al. Ese: Efficient speech recognition engine with sparse LSTM on FPGA. In *FPGA*, 2017.
[11] Yijin Guan et al. Fpga-based accelerator for long short-term memory recurrent neural networks. In *ASP-DAC*, 2017.
[12] Shuo Wang et al. C-lstm: Enabling efficient LSTM using structured compression techniques on fpgas. In *FPGA*, 2018.
[13] Xiaofan Zhang et al. High-performance video content recognition with long-term recurrent convolutional network for FPGA. In *FPL*, 2017.
[14] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.
[15] Kyunghyun Cho. From sequence modeling to translation. In *lecture note, Deep Learning for Machine Translation*, October 2015.