Efficient Methods for Mapping Neural Machine Translator on FPGAs

Qin Li, Xiaofan Zhang, *Student Member, IEEE*, Jinjun Xiong, *Senior Member, IEEE*, Wen-Mei Hwu, *Fellow, IEEE*, and Deming Chen, *Fellow, IEEE*

Abstract—Neural machine translation (NMT) is one of the most critical applications in natural language processing (NLP) with the main idea of converting text in one language to another using deep neural networks. In recent year, we have seen continuous development of NMT by integrating more emerging technologies, such as bidirectional gated recurrent units (GRU), attention mechanisms, and beam-search algorithms, for improved translation quality. However, with the increasing problem size, the real-life NMT models have become much more complicated and difficult to implement on hardware for acceleration opportunities. In this article, we aim to exploit the capability of FPGAs to deliver highly efficient implementations for real-life NMT applications. We map the inference of a large-scale NMT model with total computation of 172 GFLOP to a highly optimized high-level synthesis (HLS) IP and integrate the IP into Xilinx VCU118 FPGA platform. The model has widely used key features for NMTs, including the bidirectional GRU layer, attention mechanism, and beam search. We quantize the model to mixed-precision representation in which parameters and portions of calculations are in 16-bit half precision, and others remain as 32-bit floating-point. Compared to the float NMT implementation on FPGA, we achieve 13.1× speedup with an end-to-end performance of 22.0 GFLOPS without any accuracy degradation. Based on our knowledge, this is the first work that successfully implements a real-life end-to-end NMT model to an FPGA on board.

Index Terms—Hardware-efficient inference, neural machine translation, FPGA, high level synthesis

1 Introduction

Machine translation is one of the most popular natural language processing (NLP) tasks. Compared to traditional statistical machine translation (SMT) that relies on statistical models, neural machine translation (NMT), using DNNs (deep neural networks) to model entire input sentences and predict the likelihood of sequence of words, has demonstrated much higher accuracy [1], [2]. However, as researchers pursue the highest accuracy of NMT, the model has been aggregated with hundreds of millions of parameters and needs hundreds of GPU hours for training [3]. To reduce the model size, some researchers applied quantization mechanisms to NMT models, truncating the parameters but preserving floating-point calculations of the models due to accuracy concerns [4], [5].

Besides CPUs and GPUs, FPGA, an energy-efficient alternative, has been considered as another candidate with strong computational power and has achieved great performance on various DNN tasks [6], [7], [8]. However, the conventional ways of writing hardware description language (HDL) code for FPGAs are both painful and take much longer time than code development on CPU/GPU. High-level synthesis (HLS), converting high-level language such as C, C++ to HDL automatically, largely mitigates the time-consuming FPGA code development process [9], [10], [11], [12], [13], [14], [15].

In our work, we map the inference of a representative encoder-decoder based NMT model proposed in [16] with total computation of 172 GFLOP to a highly optimized HLS IP and integrate the IP into Xilinx VCU118 FPGA platform. The model has widely used key features for NMTs including bidirectional GRU layer, attention mechanism, and beam search algorithm. We quantize the model to mixed-precision representation in which parameters and portions of calculation are in 16-bit half precision, and others remain as 32-bit floating-point. This paper is a continuation of our previous work [17], the first real-life NMT design on FPGAs using floating-point precision. The key improvements compared to [17] include a hybrid-precision NMT model design to achieve improved board-level performance and the same level of accuracy as the floating-point version, a heterogeneous decoder design to integrate dedicated decoders for layers with different compute-to-communication (CTC) ratios, and a refined attention module to optimize the computation order and reduce process latency. To sum up, following are the contributions of this work:

- We introduce *a hardware-oriented profiler* and *a comprehensive task partitioning strategy* for mapping NMT onto FPGAs. The proposed profiler first identifies computational demands and memory requirements of the targeted NMT Please provide the author for Recommended for acceptance.model. The proposed task partition strategy then helps to allocate hardware resources for different tasks in NMT according to their compute and memory-access features, and eventually to fully utilize the available hardware resources following the profiling analysis.

- *Qin Li, Xiaofan Zhang, Wen-Mei Hwu, and Deming Chen are with the Department of Electrical and Computer Engineering, University of Illinois Urbana-Champaign, Champaign, IL 61801 USA. E-mail: {qinli2, xiaofan3, w-huw, dchen}@illinois.edu.*
- *Jinjun Xiong is with IBM T.J. Watson Research Center, NY 10598 USA. E-mail: jinjun@us.ibm.com.*

Manuscript received 1 July 2020; revised 23 Oct. 2020; accepted 30 Nov. 2020. Date of publication 25 Dec. 2020; date of current version 11 Feb. 2021. (Corresponding author: Xiaofan Zhang.) Recommended for acceptance by P. Balaji, J. Zhai, and M. Si. Digital Object Identifier no. 10.1109/TPDS.2020.3047371

- We propose *a heterogeneous decoder design* to efficiently process the decoding process of recently developed encoder-decoder NMT models. Different decoding processes within the same NMT model can be treated specifically, as we integrate two heterogeneous decoders (Decoder Five and Decoder One) to provide better accommodations of the computation demands with different CTC ratios.
- As the attention mechanism is one of the most import and compute-intensive features in NMT models, we provide *a refined attention module* to improve our design's board-level performance. It effectively increases the reuse of DNN parameters, reduces the number of external data transfer, and constructs well-balanced pipelines to maximize the performance.
- We introduce *highly optimized HLS IPs* as major building blocks for implementing the demanding NMT model on the targeted FPGA. We apply a variety of optimization techniques, such as partial on-chip weight storage, weight sharing, buffer sharing, optimized matrix-vector-multiplication (MVM), array partitioning, loop unrolling, and pipelining.
- To improve the inference speed, we leverage a *hybrid-precision model* on our board-level implementation on a Xilinx VCU118 FPGA. We quantize the NMT model based on both performance and accuracy considerations and we increase the performance over the floating-point version by 13.1× while maintaining the same accuracy.

2 Related Work

2.1 NMT

Before the wide adoption of DNNs, machine translations mainly rely on statistical methods to find the highest probability of the target language phrase when translating the current input words based on bilingual text corpora. As deep neural networks become popular, researchers start to apply fully-connected layers and recurrent neural network (RNN) layers on machine translation to provide better translation quality [18], [19]. After that, models of machine translation are mostly DNN-based with encoder-decoder structures (also known as the sequence-to-sequence models) [1], [2]. These NMT models first adopt RNNs to encode the input sentences into internal context vectors which summarize the input contexts. Decoders (which are also built of RNNs) then translate context vectors into different languages. For challenging tasks, such as long sentence translations, pure RNNs may fail to remember the previous information as the gap between the information and current task increases due to vanishing gradient. Thus, long short-term memory (LSTM), a special kind of RNN, is introduced to mitigate this issue, preserving portions of previous outputs in memory cells [20]. Another special RNN is also proposed, called gate recurrent unit (GRU), which requires less computation than LSTM while still achieving similar accuracy [21].

The next breakthrough in NMT is the attention mechanism, which helps paying "attention" to related input vectors before generating each specific output word and significantly improves the long sentence translation [16]. The encoder-decoder model with attention has become the most representative paradigm guiding the future direction of NMT model design. For example, Google proposes a large-scale NMT model, called GNMT, with 16 LSTM layers and an attention mechanism, to provide high-quality translation services [22]. Besides the RNN-based implementation, Facebook uses convolutional neural networks (CNNs) to encode sentences following the same model design paradigm [23]. Recently, a new model, called Transformer, has once again improved the translation quality [24]. It adopts the same design paradigm (the encoder-decoder with attention) but mainly depends on the attention modules (e.g., multi-head attention and masked multi-head attention) instead of the RNN-based structure. Beside the attention mechanism, major layers in this new model also share similarities with the representative one in [16], such as using feed forward layers, linear layers, and softmax layers. Therefore, we select the model in [16] for research on hardware implementation.

2.2 FPGA

We have seen extensive studies in DNN accelerator using FPGA to deliver competitive energy efficiency and performance [25], [26]. For DNN applications, matrix-vector multiplications are widely applied. Despite the wide variety of DNN models, MVM is the main computation inside each of them, such as the most distinctive attention mechanism in NMT. To accelerate the process of convolutional layers, authors in [6] propose a FPGA-based accelerator by applying loop tiling, unrolling, pipelining and data reuse under memory bandwidth and resource constraints. For LSTM, ESE is proposed in [27] to provide high-performance and load-balancing acceleration with comprehensive sparsity and quantization exploration. Other designs [28], [29] also present alternative power-efficient solutions for RNNs. Another work propose a CPU-FPGA system for NMT acceleration while only MVM engines are mapped to FPGAs, and CPU handles the rest of calculations [30]. These works focus more on building an accelerator engine solely for a single DNN layer or MVM processes, while still needing calculations on the host side to run the entire DNN model.

There are also other works that deploy a complete network on an FPGA board. For example, lots of successful computer-vision related implementations have been deployed on FPGAs [7], [8], [31], [32]. In addition, a long-term recurrent convolutional network (with both CNN and RNN) for video content recognition is mapped to a Virtex-7 FPGA with high performance and efficiency [15]. To speedup the design of FPGA-based accelerators, DNNBuilder is proposed in [8] as an automated tool to directly map customized CNNs written in software to FPGAs with automatic optimization. Besides the computer vision field, an FPGA-based RNN implementation can be used for NMT tasks, such as the design in [33]. However, the problem size of this work is below 30 GOP, which is much smaller than our real-life NMT implementation with 172 GFLOP.

Regarding our hardware implementation, we target the NMT model in [16] as it includes all essential components (especially the attention mechanism) required by modern NMT models. Starting from this model helps us better understand implementing current and future NMT workloads on FPGAs. Also, one main goal of this work is to demonstrate the flexibility and efficiency of using HLS. With the higher abstraction level of hardware design, we can make faster responses to support different models by reconstructing the behavior-level design and reusing the pre-built HLS IPs to avoid tedious and repetitive jobs. It also means we can continuously improve

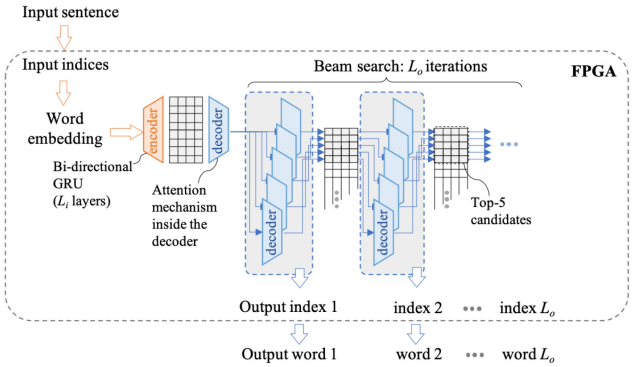


Fig. 1. Overall structure of the targeted NMT model.

the hardware design by adding new HLS IPs to support any emerging components in NMT models.

3 NMT Model

3.1 Overall Structure

The structure of the sequence-to-sequence based NMT model is presented in Fig. 1. The model is equipped with bidirectional GRU layer for encoder, and attention mechanism and beam search for decoder. Two dictionaries for source and target language respectively are used for mapping words in each language to indices. Input sentences with L_i words first are parsed to words which are then converted to corresponding indices after looking up the source dictionary. The indices are transformed to L_i embedding vectors, and then sent to the encoder. The decoder takes the encoded outputs and generates L_o output indices. The indices are transformed to words following the mapping in the target dictionary, and then detokenized to target sentences. In this project, the FPGA takes preprocessed input indices and calculates indices of target language as outputs. The maximum numbers supported for L_i and L_o are both 50, and the size of both dictionaries of source and target languages are 30000 words.

3.2 Encoder

There are two inputs for the GRU layer: a vector denoting input word and another vector representing memory of this layer. Equation (1) is the updated gate representing the fraction the memory should be updated, while Equation (2), reset gate, represents the amount of memory that should be discarded. Equations (3) and (4) generate the current output according to the gate values. The current output vector is going to be the memory for the next iteration.

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

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

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

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

A Bidirectional GRU (with output vector size of 2048) is applied as the encoder, so that both words come before and after the current word can be linked. As shown in Fig. 2, one forward GRU layer takes words in regular order and another backward GRU layer takes the words reversely. Output vectors of each layer denoting the same word are concatenated as the final encoded results. Thus, a set of

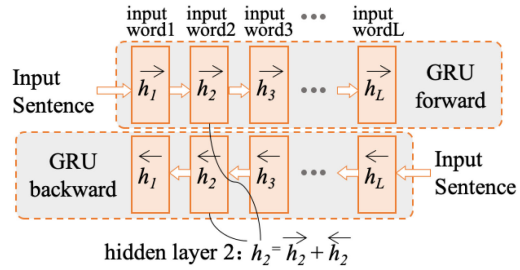


Fig. 2. NMT encoder with bidirectional GRU layer.

encoded vectors with size of $L_i \times 2048$ is computed. The encoder connects words with each other in the input sentence, making each word correlated with preceding and following words to achieve a better translation result. A FC layer with activation function $tanh$ is treated as a bridge between the encoder and decoder; it takes the average of the encoded vectors and the output is used as the initial GRU memory of the decoder.

3.3 Decoder

After the encoder finds the correlation between input words and forms a set of encoded vectors, the decoder runs recurrently and analyzes the vectors and output word in the previous iteration to predict the current output word. If the *eos* (end of sentence) is chosen as the current output word, the loop is terminated. As shown in Fig. 3, the previous output word embedding vector and encoder output are the inputs for each iteration. A GRU layer links all of the previous output words, and both GRU output (s) and encoded vectors (h_i) are sent to attention mechanism. Recurrently, the GRU output and attention output are the corresponding inputs for the next GRU layer (GRU_nl). Both of the GRU layers in the decoder have output vector size of 1024. The GRU_nl output (s) is going to be the memory portion for the first GRU layer during the next decoding iteration. The previous output word embedding vector, output of attention and GRU_nl are sent to three parallel FC layers ($FF\_Previous$, $FF\_Context$, FF_GRU) each with 512 neurons, respectively. Summations of FC outputs are sent to another FC layer with softmax function, generating a probability map representing the probability of each word in the target dictionary. The beam search algorithm determines which words to choose as the next outputs.

3.4 Attention Mechanism

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

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_i^{L_i}\exp(e_{ij})}. \tag{6}$$

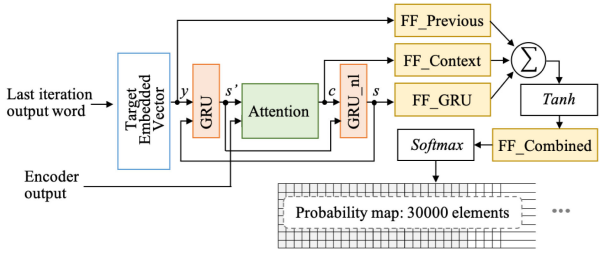


Fig. 3. Detailed decoder structure.

$$c_j = \sum_{i}^{L_i} \alpha_{ij} h_i. \tag{7}$$

The translation process is not purely one-to-one mapping. Each word in the input sentence is related to some extent to the current output words, but with different degree of relativity. The attention mechanism determines the level of relativity, paying more attention to more related input words and less to others within each decoding process. It takes the last output word vector and the encoded input vectors as inputs, computing the importance of each encoded input indices in terms of generating the next output word. We use Bahdanau Attention in Equations 5 $\sim$ 7 in the NMT model [16]. In each iteration j, the energy state (denoting the level of relativity) of each of L_i encoded vectors (e_{ij}) is calculated taking the encoded vector h_i and GRU output s'_j. Then the level of importance for each encoded vector α_{ij} is computed by performing softmax of L_i energy states. The final attention outputs for the current decoding iteration (c_j) are weighted average of the encoded vectors.

3.5 Beam Search

In each decoding process, a probability map is generated denoting the probability of being the next potential output word index. Choosing the highest probability for a complete sentence with multiple words requires exponential decoding which is unrealistic, while only preserving the highest probability word index does not guarantee the optimal results. To overcome this challenge, beam search algorithm is introduced, which only keeps a fixed number of active candidates at each time step. We choose beam size $K = 5$. Thus, after the initial index *BOS* (begin of sentence) is sent to decoder and generates a probability map, only the top five word indices are preserved. Then after the first iteration, each iteration only has at most five decoding processes as shown in Fig. 4, and among the 5×30000 scores generated, the next top five potential answers are preserved. The translation process terminates when all five tracks reach *eos*, where the sentence with the highest score is selected.

4 Design Approaches and Profiling

Unlike instruction-based CPUs and GPUs, FPGA allows direct computing and data transferring on hardware and the parallelism of each layer is adjustable. However, each FPGA has limited resources and memory bandwidth. Thus, profiling the NMT model is needed to properly partition the resource to different computational and memory-related tasks. In this section, we present our profiling results and analyze the computational demand and memory overhead of the NMT model.

4.1 Design Approaches to FPGA Implementation

To implement the targeted NMT on FPGAs, we cover certain design approaches, including model profiling, task partitioning, and HLS IP configuration, and the goal is to generate the optimized HLS-based designs by considering the complexity of input models and available resources of the targeted devices. To achieve this goal, the proposed partitioning strategy first takes the profiling results of the targeted NMT model (e.g., layer type, compute/memory demand, CTC ratio, and operation distribution) as inputs and selects the corresponding HLS IPs for constructing the whole design. These HLS IPs

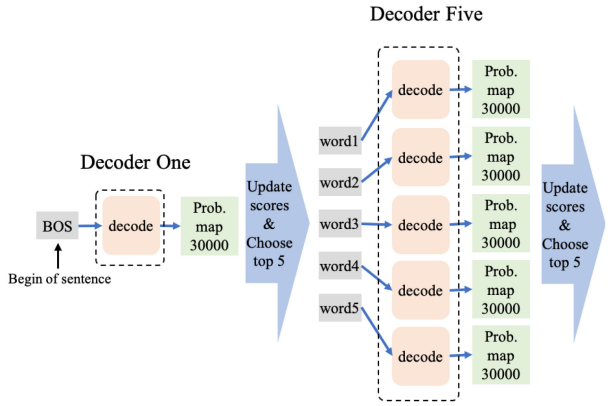


Fig. 4. Beam search with two kinds of decoding iterations.

(such as the MVM kernels, non-linear functions, sorting kernels) are the basic hardware building blocks, which are highly configurable and can be configured to deliver different performances by consuming more or fewer resources. The next approach is to configure these selected HLS IPs properly to maximize the achievable performance given limited hardware resources. Since most of the computations in NMT can be represented as a nested for-loop style, we adopt HLS loop-optimizations (e.g., loop unrolling and loop pipelining) to speed-up corresponding IPs by increasing their parallelism. Now, the partitioning strategy has finished its job by delivering HLS IPs with proper configurations. After that, these configured HLS IPs are passed to the HLS design flow for RTL generation. We eventually have the board-level implementation after logic synthesis, placement, and routing for the targeted FPGA.

These design approaches also show good versatility, and they can be applied to handle other NMT models or DNNs once we have the corresponding configurable HLS IPs available. Since most of the DNN use modular designs (e.g., repeatedly using the limited types of neural network layers to construct the entire DNN), they can be well accommodated by using our approaches, starting from model profiling, task partitioning, to hardware implementation using HLS IPs.

4.2 Computational Demand

The targeted NMT model requires 172 GFLOP total computation and its computational distribution translating 50-word source sentence to 50-word output text is shown in Fig. 5. The encoding process (FC bridge included) consists of 50 steps of bi-directional GRU layer with 2048 neurons in total and an FC layer with 1024 neurons. However, even with large number of neurons, the encoder is negligible (0.43 percent of total computation) compared to the total decoding processes since the beam search algorithm requires five decoding processes per step, and there are 50 steps in total. For each decoding process, attention mechanism has the most computation (90.64 percent of the decoding process). The second most computation-demanded is the final *FF_Combined* layer since it has 30000 neurons in order to generate the probability map. However, the amount of computation in this FF layer is still minimal compared to the attention mechanism. Thus, an efficient optimization technique applied on attention mechanism layer can significantly improve the total NMT performance.

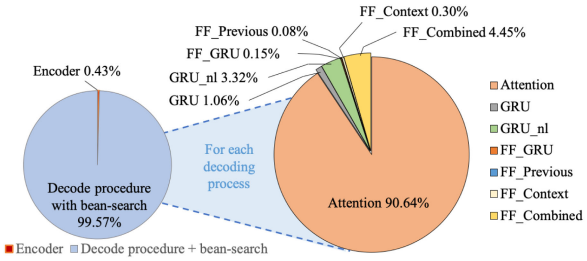


Fig. 5. Operation distribution in the targeted NMT (left) and computational demand breakdown in a single decoding process (right).

4.3 Memory Overhead

With limited storage, parameters of the NMT model cannot be fully loaded on-chip. Thus, most of them are stored off-chip, and load from off-chip DRAM whenever they are needed. For optimization, memory overhead should be analyzed in order to properly arrange the computation order and choose the portion of parameters to store on-chip. Parameters in the NMT model are weights, biases and word embedding vectors. Weights are always used in dominated matrix-vector-multiplication operations, while unlike convolutional layers, each element in the weight is only used once during MVM process. Thus the computation-to-communication (CTC) ratio is only one, making the process memory-bounded. The computation can only proceed after the weight is loaded to the buffer. However, due to beam search algorithm, if the five decoding processes in the same step run concurrently, even though they aim at different output sentences, they can potentially share the loaded weights and compute correspondingly, increasing the CTC ratio to five. In addition, when we calculate energy states in attention mechanism, all encoded vectors actually use the same set of weights, so that the weights are highly reused. Time could be wasted if we load those weights iteratively during each energy state calculation.

5 NMT Hardware Design

The profiling analysis points out the dominant computational demand and memory overhead of the targeted NMT model, so that we can configure our hardware accordingly. Additionally, we further improve the translator with highly optimized IPs, quantization technique and algorithm refinement to achieve optimal performance. In this section, we demonstrate our design methodology regarding the complicated network interconnection, computational demand, and resource utilization using high-level synthesis (HLS).

5.1 NMT Overall Structure

Overall accelerator structure of NMT is shown in Fig. 6. Our design efficiently utilize the resources on board using customized modules. The NMT hardware consists of three portions: logic, on-chip memory and off-chip memory. The logic portion handles all of the control logic and computation, which includes LUT (lookup table), FF (flip-flop) and DSP resources. BRAM is the on-chip memory while DRAM is the off-chip memory.

In NMT logic portion, the translator consists of an encoder and an decoder. According to the profiling results mentioned in the previous section, computation of the encoder is negligible compared to decoders. Therefore, most resources are allocated for decoder optimization. Two kinds of decoder modules, *Decoder One* and *Decoder Five*, are instantiated, where the former is only used for the first iteration of the decoding process, and the latter is for decoding five different potential tracks. Therefore, three main modules are instantiated in the hardware, the encoder with bidirectional GRU layer, Decoder One for decoding one process, and Decoder Five for decoding five processes. Inside those main modules, MVM kernels as well as non-linear computation function modules are created, where each MVM Five module handles five matrix-vector-multiplication processes while MVM One only handles a single process. For the encoder and Decoder One, only single MVM One kernel module is shared by all of MVM processes, while multiple MVM Five kernels are instantiated in the most computationally demanded Decoder Five module. As shown in Fig. 6, each set of non-linear and MVM Five modules is instantiated for computation in GRU, attention mechanism, GRU_nl layers. For all fully-connected layers in Decoder Five, one MVM Five module is shared by all of them. Besides those main modules, logic for sorting and selecting the best five scores is also included.

For on-chip memory, we create buffers on the BRAM for intermediate results, MVM processes and scores of sentences. Partial weights and attention-related buffer, Att_wh, are stored on the URAM. Other parameters including weights, biases, word embedding vectors for both source language and target language, input indices and output indices are stored off-chip.

5.2 Heterogeneous Decoders

Based on the beam search algorithm in Fig. 4, two kinds of decoding processes are executed during translation. We separate them into different modules, Decoder Five and Decoder One, for two reasons. First, when decoding five processes, weights can be shared by them if they run concurrently. Encapsulating them into a single module can assure that each MVM process shares the weights. Second, there are redundancies in the model, so that the first decoding process can calculate certain results and store them in a buffer, where the rest of decoding process can directly use without redundant calculation.

5.2.1 *Decoder Five Versus Decoder One*

Decoder Five is responsible for most of calculations in NMT. Five decoding processes run concurrently with shared buffers and computational IPs. As we apply beam search algorithm with beam size 5 in our model, five decoding processes are needed for each iteration in the decoder, and those decoding processes require the same set of weights and biases. Instead of executing those processes sequentially, the processes are running concurrently, so that the weights loaded from off-chip memory by one process can be reused by the other four decoding processes, increasing the computation-to-communication (CTC) ratio to five. Thus memory-bounded MVM operations can use more CEs (MVM Five kernel) for computation achieving better performance. Compared to Decoder Five, Decoder One only needs to handle one input vectors for only one iteration, a single MVM kernel is enough to handle all of the decoding task.

5.2.2 *Refined Attention Mechanism*

To further optimize our design, we change the computation order and store the pre-calculated results in the attention

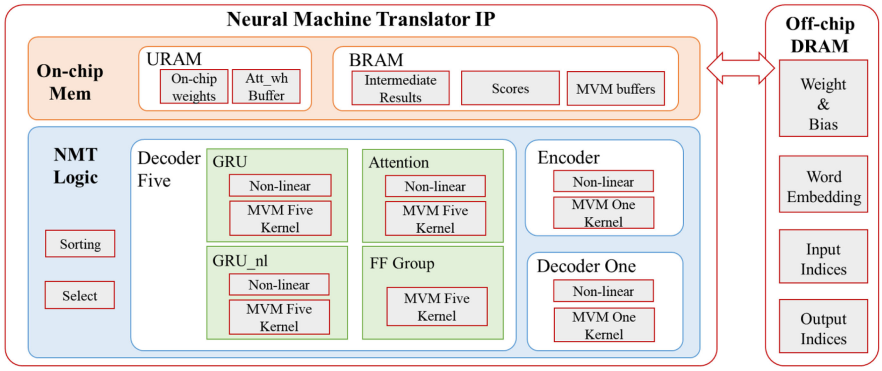


Fig. 6. Accelerator-level structure of NMT.

mechanism module. The optimization technique greatly reduces redundant calculations, so that the calculation is first done and results are stored by Decoder One for re-usage in Decoder Five.

Among all of the calculations, attention mechanism accounts for 90 percent of them according to our profiling results shown in Fig. 5. The large amount of calculation is used for computing the energy state of each of L_i encoded vectors, e_{ij} in Equation (5), which takes one of encoded vector h_i and GRU output s'_j as inputs. During each decoding iteration j, L_i energy states need to be calculated where two MVM processes are involved in this calculation, $U_a s'_j$ and $W_a h_i$ with weight size of 1024×2048 and 2048×2048 respectively. However, for each encoder iteration i, $U_a s'_j$ is constant and $W_a h_i$ is independent of each decoder iteration j. It is unnecessary to calculate them in a loop during every decoding process.

Our optimized attention module is shown in Fig. 7. In our design, $U_a s'_j$ is first calculated and stored temporarily for both MVM calculations in Decoder One and Decoder Five. $W_a h_i$ is only calculated in Decoder One. The results of each encoded vectors are stored in a buffer so that the Decoder Five can directly use them during calculation. In this way, there is no MVM calculated iteratively in attention mechanism after the first decoding process. In addition, since attention mechanism dominates the decoder calculation, on-chip weight storage can be applied to this module, so that when calculating the energy states for each encoded vector, we do not need to iteratively load the weights from

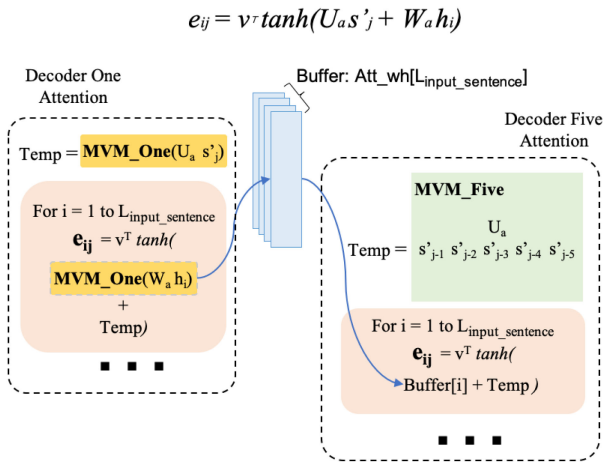


Fig. 7. Optimized flow for attention mechanism.

off-chip DRAM, while other on-chip weights can also be used for quicker attention mechanism calculation.

5.3 Matrix-Vector Multiplication

According to the equations for FC layers, GRU layers and attention mechanism, matrix-vector multiplication (MVM) is the main computation inside the NMT model, in which the vector represents input and matrix represents weights. We utilize the benefits of half-precision data format with pipelined compute engines to optimize the MVM process.

5.3.1 Half Precision

Half precision is a 16-bit floating-point format, which consists of 1 sign bit, 5 bits for exponent and 11 bits for fraction. Compared to 32-bit single-floating-point precision, 16-bit half precision has less memory and computational resource requirements. As shown in Table 1, fmul and fadd represent multiplication and addition for float values, while hmul and hadd stand for multiplication and addition of half values. The DSP, flip-flop, and lookup table usage of float calculation are 2-3 times more than that of half calculation.

However, half precision has much more range and precision limitations. To avoid out-of-range issues and maintain the accuracy, while still utilizing the benefits from the half-precision datatype, the NMT model is quantized with mixed-precision representation. In our quantization scheme, all of the parameters including weights, biases and word-embedding vectors are represented by 16-bit half precision to reduce the overall loading workload. Buffers and computational IPs for MVM kernels are also half-precision which require much less utilization than the floating-point alternative. Other non-linear operations and results remain as 32-bit floating-point to prevent from overflow or underflow. Under limited off-chip memory bandwidth, twice the weights can be loaded per unit time, while similar accuracy can be achieved by using the mixed-precision method after readjusting the weights arrangement by retraining.

5.3.2 MVM Kernel Design

Two kinds of MVM processes happened in the translator, $MVMOne$ and $MVMFive$, where the former takes one input vector while the latter takes five input vectors. In Fig. 8, MVM Five is presented to demonstrate our MVM kernel design.

To fully utilize the 512-bit data width in the AXI (Advanced eXtensible Interface) bus which is used for transferring off-chip memory data to the NMT IP, we set the

TABLE 1
Utilization Comparison Between Half and Float Precision Calculation

Precision	Float32		Half16	
Calculation	fadd	fmul	hadd	hmul
DSP	2	3	2	2
FF	177	128	94	64
LUT	226	77	111	33

TABLE 2
MVM Kernel Comparison With Different Parallel Setups

MVM Kernel	Parallelism	Norm. Performance	DSP
1 × CE32	32	0.42	160
2 × CE32	64	0.81	320
4 × CE32	128	1.00	640
1 × CE64	64	0.51	320
2 × CE64	128	1.00	640
4 × CE64	256	1.00	1280

width of off-chip data port to 512 by specifying the port type as *ap_int* $< 512 >$, a special data format in HLS with customizable data width. As shown in Fig. 8, 32 weights in half precision are copied from off-chip DDR memory to MVM buffer per loading time. Besides weights, portions of input vectors are converted from floats to half values and also copied to MVM buffers.

After the loading process, data in the MVM buffer are sent to a MVM kernel inside which multiple compute engines (CE) are instantiated. Each of them is a MAC (multiplier–accumulator) unit which receives n pairs of half values, first multiplies each pair, and adds all of the results using an adder tree. Each computation stage in the CE is pipelined so that multiple executions in one CE can be overlapped. For instance, if we need to compute two sets of inputs, once the first set finishes the calculation of the multiplication stage, the next one can start to use the CE. The next calculation does not have to wait until the current calculation finishes. After computing portions of inputs, the results are converted back to float and added to corresponding portions of the output vectors.

The load, compute, and store processes are pipelined using pragmas and the global MVM buffers are fully partitioned for optimal performance. Sizes of global buffers and compute engines are fine-tuned so that the loading time and computation plus storing time can be highly matched so the runtime of these processes can be mostly overlapped in the pipeline to achieve the best performance.

5.3.3 MVM One Versus MVM Five

In pipelined load, compute and store processes, different numbers of MVM buffers and CEs are instantiated for MVM One (used for encoder and Decoder One) and MVM Five (used for Decoder Five) in order to compensate the dominant loading time with matched computational resources. Our goal is to construct a balanced pipeline, where each stage should has similar latency, so that we can maximize the pipeline performance. To adjust the stage latency, we adopt the proposed partitioning strategy to provide guidelines for parallel setups. In our design, each HLS IP supports multi-dimensional parallel processing, which enable flexible adjustments. For example, we list different configurations of the proposed MVM Five kernel in Table 2, when targeting the Float32 NMT model. We notice different parallelism setups not only affect resource consumption but performance. Also, allocating more DSP resources does not mean better performance. The MVM Five kernel with 4 × CE64 achieves the same performance as the case with 2 × CE64. The reason is that the data loading latency starts dominating the overall latency once the parallelism is higher than 2 × CE64 in this particular scenario. Building even bigger computation units can not help improve performance. Therefore, we need the task partitioning strategy to help manage resource allocations and deliver HLS IPs with suitable configurations for optimized results.

Similarly, when targeting the hybrid-precision NMT model, we decide to use CE64 in our MVM design after model profiling and task partitioning. It means each CE receives 64 pairs of half values. The weight buffer size of both MVM designs is 64×2. Thus, each time 64 weights from neighboring 2 rows are loaded off-chip. On the other hand, with CTC = 5, three CEs are instantiated in the MVM Five kernel, while only one CE is instantiated for MVM One kernel with CTC = 1. More computational resources are allocated for MVM Five to match the higher CTC value. A 64×5 buffer is created in MVM Five, while a smaller 64×1 buffer is used for MVM One kernel to load input vectors of one or five decoding processes.

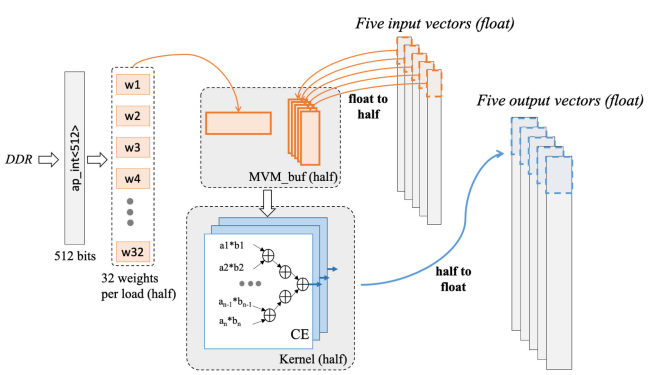


Fig. 8. MVM five kernel design with mixed precision in decoder five.

5.3.4 Float MVM Versus Mixed-Precision MVM

Under memory-constraint situation, parameters represented by half precision can load double parameters per unit time compared to the float parameters. A simplified timing diagram comparison between float and mixed-precision MVM Five design is shown in Fig. 9. Even though the mixed-precision MVM model adds type-conversion operations, the additional time is hidden by the pipeline. The dominant operation is still the loading process. For float MVM, the pipeline factor $ii = 8$, where ii represents initiation interval, the number of clock cycle from the beginning of current iteration until the next iteration can start. Therefore, for float MVM, it takes 8 cycles to load all of the weights to the on-chip MVM buffer. However, for pipelined mixed-precision MVM, $ii = 4$, proving that half precision can double the loading speed and thus under memory bounded situation has 2x speedup.

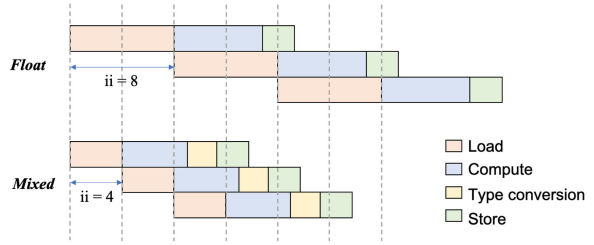


Fig. 9. Timing diagram comparison between float MVM and mixed-precision MVM.

5.4 Optimization Techniques for HLS IP Designs

Due to limited resources on the FPGA board, on-chip memory cannot hold all of the parameters and intermediate results. Therefore, inputs and weights need to be loaded from off-chip DDR memory. In addition, most of operations are memory-bounded MVM operations, and speed of loading data is limited. Under both resource and memory bandwidth constraints on the FPGA board, general optimization techniques are applied to all of modules to accelerate the NMT with affordable resource overhead. As an example, the structure of Decoder Five utilizing all of the general optimization techniques is shown in Fig. 10.

5.4.1 Compute Optimization and Memory Management

Loop Unrolling and Pipelining. Conventionally, iterations of loops in C programs are executed sequentially and execution of the next iteration needs to wait until the current loop has completed. By adding loop unrolling and pipelining pragmas in a loop, more parallelism can be exploited by HLS. Multiple iterations can either run completely in parallel or concurrently with partial overlapped runtime. However, dependency due to same-array access might make the optimization invalid. To solve this issue, pragmas for array partitioning are added to partition the array into different memory blocks so that multiple array elements can be accessed simultaneously. We applied loop partitioning in all of our buffers to store intermediate results, and loop unrolling on the non-linear function calculations.

Buffer Sharing. Due to limited on-chip memory blocks, some memory blocks for storing temporary intermediate results are shared. For example, the two GRU layers with the same number of neurons in the decoder share the intermediate update gate and reset gate results, and the same situation happens for the forward and backward layers of the encoder. Similarly, all MVM kernels in the decoder share the same set of global buffers.

IP Sharing. Each layer contains multiple MVM processes and non-linear calculations. MVM kernels and non-linear units are shared by a single layer or multiple layers. The degree of IP sharing is carefully tuned, because the model cannot be fully mapped into FPGA without sharing, while too much sharing may cause enormous creations of MUXes and failure in timing requirements due to long critical paths. For example, in Fig. 10, one MVM kernel is instantiated in *GRU, attention mechanism*, and *GRU_nl* respectively, while a single MVM kernel is shared by four FF layers represented as *FF Group*. For the encoder and Decoder One, a single MVM kernel is shared by the entire module.

5.4.2 Examples of the Optimization Techniques

To better explain the optimization techniques, we use an MVM module as an example in Algorithm 1, which performs matrix-vector-multiplication for a 512x512 matrix and a 512-dim vector input and generates a 512-dim vector. In line 1, we specify the interface of this module. Since DNN parameters are loaded from DRAM (external memory), we point the argument named `data` to the AXI bus interface (line 2), and we use BRAM (on-chip memory) to keep the input and output vectors.

To reduce data loading latency, we attempt to load more than one element from the 512-dim input vector. Therefore, we use a loop unrolling (line 8) directive to unroll the inner for-loop (line 9) by a factor of eight. In this case, eight input elements are fetched simultaneously every clock cycle and stored into a local array variable called `kvector`. To support such parallel data loading, we partition the `in[512]` to split it into eight equally sized blocks interleaving the original array elements. Only in this case can we have eight individual data blocks for fetching eight data simultaneously. The unrolling and partitioning factors need to be the same, and they can be configured to other numbers to increase or decrease the data loading parallelism.

After loading inputs from on-chip buffers (line 7∼10) and external memory (line 13∼20), the MVM module starts working on computations. Inside the kennel module (line 21), we instantiate two CE64 modules (`K_NUM = 2`), and each of them takes 64 inputs (`K_SIZE = 64`) and performs multiply and accumulation and generates a 64-dim vector as a result (line 22∼23). With this configuration, the total parallel factor of this MVM module is 2×64 = 128. Since we also apply loop pipelining (line 12), critical stages, including data loading, compute, type conversion, and result store, are executed in a pipelined manner, as shown in Fig. 9.

5.5 High-Level Synthesis

HLS offers a higher level of design abstraction, which can significantly improve the productivity of FPGA design and

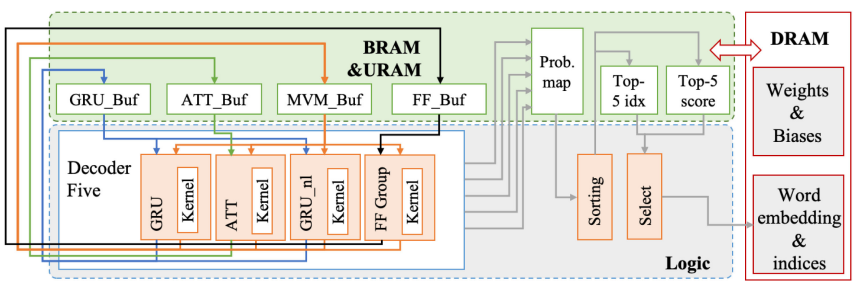


Fig. 10. Detailed structure of Decoder Five and other related logic.

Algorithm 1. The Optimized MVM Design

```
1: void mvm(ap_int<512> *data, float in[512],
   float out[512]){
2: #pragma HLS INTERFACE m_axi port=data
3: #pragma HLS ARRAY_PARTITION variable=in cyclic
   dim=1 factor=8
4: #pragma HLS ARRAY_PARTITION variable=out
   cyclic dim=1 factor=8
5:  int i,j,k,l,kk; int K_SIZE=64; int K_NUM=2;
6:  for(k=0; k<512; k+=K_SIZE){
7:    for(i=0; i<K_SIZE; i++){ //To fetch
     intermediate results
8: #pragma HLS UNROLL factor=8 //parallel data
   loading
9:      kvector[i]=(half) in[i+k];
10:   }
11:   for(l=0; l<512; l+=K_NUM){ //To fetch
     parameters from DDR
12: #pragma HLS pipeline ii=4 //pipelined the
   loading/compute/store
13:     for(j=0; j<K_SIZE; j+=32){
14:       for(kk=0; kk<32; kk++){
15:         kmatrix[j+kk][0]=Trans_half(data[...]);
16:     } }
17:     for(j=0; j<K_SIZE; j+=32){
18:       for(kk=0; kk<32; kk++){
19:         kmatrix[j+kk][1]=Trans_half(data[...]);
20:     } }
21:     kernel(kmatrix, kvector,kout);
22:     out[l]+=(float)kout[0];
23:     out[l+1]+=(float)kout[1];
24: } } }
```

support efficient design space exploration, especially for a large-scale design, such as our targeted NMT model with total computation of 172 GFLOP. In order to design a translator using HLS, we need to first write the model to synthesizable C/C++ version, removing dynamic allocations and recursions. Functions and sub-functions are created for each layer representing hardware modules. Fixed-length arrays are allocated as on-chip memory blocks for storing intermediate and final results, and various operations are converted to specialized logic clusters. we add specific pragmas to notify the HLS compiler for special optimizations, such as loop unrolling and pipelining. We also create global buffers for data sharing between modules, and design reusable optimized HLS IPs for each module to achieve the best performance. HLS tools will then finish scheduling, binding, and generating RTL designs with cycle-accurate descriptions. Since the RTL codes are not directly written by designers, the design performance largely depends on the quality of HLS codes and whether the pragmas can be used accurately. Since HLS is not good at describing control logic and memory management, improper use of the HLS code can cause lower performance or efficiency.

6 System-Level Architecture

6.1 HLS IP Integration

After the development of an HLS IP for our NMT model, a new AXI4-Lite port is added for revealing the translator's control signals, which are memory-mapped from AXI4-Lite interface to DMA (direct memory access). Among the control signals, *ap_start* is used to start the HLS IP, *ap_idle* denotes that the IP is in idle state.

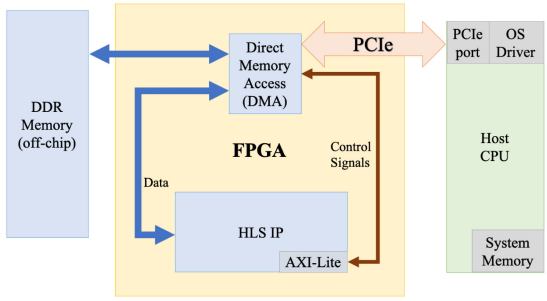


Fig. 11. System-level architecture for the NMT model.

The overall architecture design is shown in Fig. 11. The host is a desktop PC with an Intel Core i9-9900K CPU, and the VCU118 FPGA is connected with the host through a PCIe 3.0 interface. In our experiment, the host CPU is only responsible for passing the input sentences and operating signals, while the FPGA handles all the computations. We then apply a memory-map based mechanism to complete the handshake between the FPGA board (the translator side) and the host CPU (host side). The overall architecture includes off-chip DDR memory, DMA, HLS translator IP and the CPU on the host side. DMA controls the dataflow among CPU, DDR, and the translator. Host CPU sends inputs, passes commands to activate the translator and receives outputs. A driver on the host side is installed to enable communication between DMA and host CPU through PCIe.

6.2 Off-Chip Data Placement

The off-chip DRAM is partitioned to dynamic input/output regions as well as static regions for NMT parameters. As shown in Fig. 12, input and output regions include input and output word indices and the length of input and output sentences respectively. Static parameters regions include source and target word embedding vectors as well as weights. Before the translation process, the pre-trained weights and biases are rearranged following the computation order on the host side. All of the parameters are initially half-precision values. They are concatenated to 512-bit format, using $ap\_int<512>$ data type. The reordered data is sent to static region on DRAM memory of the FPGA board through PCIe port. By appying those modifications, we can activate DDR burst mode and maximize the memory-bandwidth occupation during translation. Once the host program is launched, the input sentences are parsed to tokens and then mapped to word indices. The indices as well as sentence lengths are concatenated to $ap\_int<512>$ values, and sent to a specific region in the DRAM. After inputs are ready, the host program launches the HLS IP and both concatenated output size and output indices are written to a specific region so that the host program can get the translation result.

7 Experimental Results

7.1 Prepare Work

We use Xilinx Vivado Design Suite v2019.2 to implement our HLS IP, integrate the IP into Vivado block design, synthesize the netlist, perform placement and routing, and

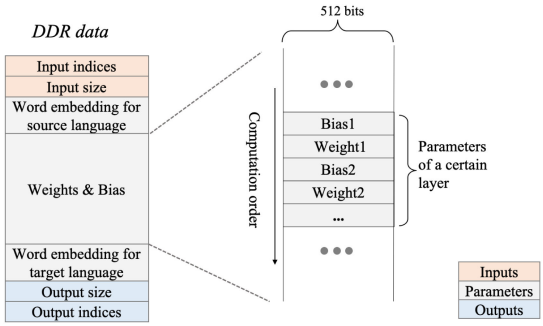


Fig. 12. Data placement on off-chip DRAM.

finally generate bitstream for UltraScale+ VCU118 FPGA development board. The degree of loop unrolling and array partitioning in the HLS IP is fine-tuned to prevent from congestion issues during placement and routing. In order to get weights for our NMT inference, we use a Pytorch implementation of the model in an open-source project named NJUNMT-pytorch.[1] English-to-French translation is chosen to demonstrate our work by training the model using English-French sentence pairs in Europarl dataset [34].

7.2 Accuracy

We use PyTorch 1.3.1 to train both half and float weights for the targeted NMT model. As shown in Table 3, PyTorch Float32 denotes the floating-point implementation in PyTorch while PyTorch Half16 represents the same model with half-precision weights. To verify the functional correctness of our FPGA design, we first implement the Float32 model on FPGA and compare the outputs to the same model running by PyTorch in a separate CPU. Since the results are identical, we can demonstrate that our FPGA design is functionally correct. We then calculate the BLEU score (which is a standard set of criteria to evaluate the quality of text [35]) of these models listed in Table 3 when running on WMT14 (Conference on Machine Translation) English-French test dataset. As shown in Table 3, we get slightly better accuracy result due to regularization effect with less number of bits. All of the three results are similar proving that there is no accuracy degradation in our work even though partial calculation is in half precision.

It is possible to have more aggressive quantization schemes, using eight or even fewer bits for hardware implementation. Since the proposed HLS IPs are highly configurable, we can adapt to arbitrary quantization schemes for inputs, outputs, and intermediate results. However, we did not involve more aggressive quantizations in the proposed NMT design due to accuracy concerns.

7.3 Performance and Utilization

Multiple NMT designs are compared in Table 4. The utilization is presented as a set of percentages using the same FPGA (VCU118), and 50-word Translation means the time needed for translating a 50-word English sentence to a French sentence. We list four models, *Float*, *Orig*, *Onchip*, and *Optimized*, where *Float* represents the original float NMT model implemented in [17], *Orig* is direct conversion from the original float model to mixed-precision model without additional optimization, *Onchip* implementation adds on-chip weight storage for the attention mechanism, and *Optimized* represents our current design with mixed-precision MVM kernels, on-chip attention weight storage and refined attention mechanism.

1. https://github.com/whr94621/NJUNMT-pytorch

TABLE 3
Accuracy Comparison on WMT English → French (newstest2014)

Model	PyTorch Float32	PyTorch Half16	our work
BLEU	22.3	21.9	22.6

7.3.1 Float Versus Half

Since half-precision weights and half-precision calculation require less memory and computational resource, more parallelism of loop unrolling and array partitioning can be applied to the design. Also, twice as many parameters can be loaded per unit time when parameters are represented as half values. As shown in Table 4, we achieve 2.5x speedup compared to the float model after transforming to mixed-precision model with fine-tuned parallelism. The float model uses much more LUT and FF than Half implementations, making the routing process much harder. Partial intermediate results are stored into URAM instead of BRAM due to congestion. We attempt to add the optimization techniques similar to those we used in the mixed-precision model, but the float model always fails due to congestion. The on-chip memory left is also not enough for the entire attention module.

7.3.2 Half NMT Implementations

The profiling result proves that the attention mechanism accounts for most of the calculation in NMT. Therefore, we statically stored the attention parameters on-chip, and also added some parallelism in calculating non-linear functions. Compared to *Orig*, the latency of *Onchip* is reduced from 41.7 s to 19.5 s, and 40 percent more URAM usage is introduced. The translation latency is cut to 7.86 s after keeping the parallelism level the same as *Orig* and applying attention mechanism refinement. In terms of utilization, besides on-chip memory usage, three half NMT implementations use similar amounts of resource because MVM kernels are shared inside modules, and utilization cannot be reduced even if the total number of calculation is less than before.

7.4 Comparison to FPGA-Based Designs

In the previous NMT work [17], the end-to-end performance is based on HLS estimation while the real board-level result is different since the HLS tool cannot estimate off-chip data transfer latency through AXI bus. It can only process the floating-point model, which requires twice as much on-chip memory capacity as the proposed design to store the same amount of data. Besides, the resource overheads of multiplication using Float32 and Half16 are quite different, as shown in Table 1, especially for LUT and FF. With respectively 22 and 11 percent higher utilization of LUT and FF compared to the *Optimized* design, the router fails to properly resolve the congestion issue in the previous design, so we have to lower the level of parallelism for the design in order to map it onto the FPGA board. On the contrary, our proposed design adopts the hybrid-precision scheme that helps significantly improve the board-level performance.

We present the comparison results in Table 5 among the proposed design and the FPGA-based designs published in

TABLE 4
Utilization and Performance of the FPGA-Based NMT Implementations

Model	Float [17]	Orig	Onchip	Optimized
Precision	Float32	Mixed with Float32 & Half16		
LUT	69%	47%	50%	47%
FF	33%	22%	22%	22%
BRAM	63%	62%	61%	67%
URAM	16%	16%	57%	41%
DSP	70%	70%	76%	71%
50-word Translation	103s	41.7s	19.5s	7.86s

[33] and [17]. We calculate the end-to-end performance by the information provided in [33] (problem set size: 2.76 mega-operations divide by overall latency: 0.39 second). As shown in Table 5, our work has much better performance than previous works with our *Optimized* implementation. Compared to our previous NMT work [17], we achieve 13.1× speedup after all of our optimization techniques with end-to-end performance of 22.0 GFLOPS.

Compared to the design in [17], we achieve a huge performance gain because the optimized HLS IPs are able to deliver a higher-level of parallelism even with the same FPGA. Such a improvement also comes from the proposed task partitioning by providing suitable IP configurations (e.g., buffer size, parallelism factors, number of CE instances, etc.) and the heterogeneous decoder and the refined attention module by reducing the number of data transfers from the external memory and increasing the parameter reuse opportunity.

7.5 Comparison to CPU and GPU Designs

We extend our comparison to evaluate the performance and efficiency of the CPU- and GPU-based implementation when targeting the same NMT model with problem size as 172 GFLOP. By running the same 50-word sentence translation task, we list the results in Table 6. Both CPU and GPU designs use PyTorch 1.3.1 to execute the targeted model and both of them target the Float32 version, while the proposed FPGA design targets the hybrid-precision model. In this case, the GPU-based design achieves the best performance and completes the translation in the shortest time. Our design in VCU118 delivers the highest efficiency compared to the other approaches.

8 Conclusion

In this work, we mapped a 172 GFLOP Neural Machine Translation model with mixed-precision representation to a single FPGA board. We cut the redundant calculation in NMT model by changing the calculation order and storing pre-calculated results appropriately. The proposed design achieved much better performance by applying optimization techniques, such as partial on-chip weight storage, weight sharing, buffer sharing, optimized matrix-vector-multiplication IPs, array partitioning, loop unrolling and pipelining. Unlike previous works which played with small-scale DNN models, we implemented a real-life NMT model with all the latest features including bidirectional GRU, attention mechanism, and beam search algorithm. By taking advantage of HLS, we fine-tuned our NMT design with much less effort than using traditional HDL to achieve optimal performance under FPGA resource constraints.

TABLE 5
Comparison to Previous FPGA-Based Designs

Reference	[33]	Float NMT [17]	our work
Targeted Model	3 LSTM	NMT	NMT
Total Size	2.76MFLOP	172GFLOP	172GFLOP
FPGA type	Virtex-7	VCU118	VCU118
Precision	Float32	Float32	Float&Half
DSP Usage	1176	4791	4838
Frequency	150MHz	100MHz	100MHz
End-to-end perf.	0.007 GFLOPS	1.68 GFLOPS	22.0 GFLOPS

TABLE 6
Comparison to CPU- and GPU-Based Designs

Platform	CPU Intel i5	GPU NVidia RTX2070	FPGA (our work) Xilinx VCU118
Frequency	3.1 GHz	1.4 GHz	100 MHz
Precision	Float32	Float32	Float32 & Half16
Latency (s)	9.68	4.90	7.86
Performance (GFLOPS)	17.8	35.1	22.0
Efficiency (GFLOPS/Watt)	0.19	0.20	1.05

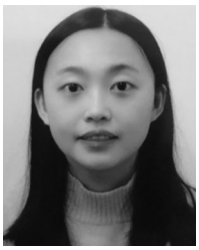
Acknowledgments

This work was supported in part by the IBM-Illinois Center for Cognitive Computing System Research (C^3SR) – A research collaboration as part of the IBM AI Horizons Network, and a Google PhD Fellowship to Xiaofan Zhang. Qin Li and Xiaofan Zhang contributed equally to this work.

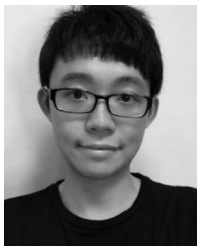
References

[1] N. Kalchbrenner and P. Blunsom, "Recurrent continuous translation models," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2013, pp. 1700–1709.
[2] I. Sutskever *et al.*, "Sequence to sequence learning with neural networks," in *Proc. 27th Int. Conf. Neural Inf. Process. Syst.*, 2014, pp. 3104–3112.
[3] M. X. Chen *et al.*, "The best of both worlds: Combining recent advances in neural machine translation," in *Proc. 56th Annu. Meeting Assoc. Comput. Linguistics*, 2018, pp. 76–86.
[4] J. Quinn and M. Ballesteros, "Pieces of eight: 8-bit neural machine translation," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Human Lang. Technol.*, 2018, pp. 114–120.
[5] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik, "Neural network distiller: A Python package for DNN compression research," 2019, *arXiv: 1910.12232*.
[6] C. Zhang *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2015, pp. 161–170.
[7] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 26–35.
[8] X. Zhang *et al.*, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proc. Int. Conf. Comput.-Aided Des.*, 2018, Art. no. 56.
[9] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "xPilot: A platform-based behavioral synthesis system," in *Proc. SRC Techcon. Conf.*, 2005.
[10] Xilinx, "Vivado high-level synthesis," Accessed: Oct. 23, 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[11] D. Chen, J. Cong, Y. Fan, and L. Wan, "LOPASS: A low-power architectural synthesis system for FPGAs with interconnect estimation and optimization," *IEEE Trans. Very Large Scale Integr.*, vol. 18, no. 4, pp. 564–577, Apr. 2010.
[12] A. Canis *et al.*, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proc. 19th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2011, pp. 33–36.

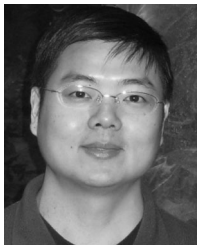
[13] X. Liu *et al.*, "High level synthesis of complex applications: An H. 264 video decoder," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 224–233.
[14] Y. Guan *et al.*, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *Proc. IEEE 25th Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2017, pp. 152–159.
[15] X. Zhang *et al.*, "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *Proc. 27th Int. Conf. Field Programmable Logic Appl.*, 2017, pp. 1–4.
[16] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014, *arXiv:1409.0473*.
[17] Q. Li *et al.*, "Implementing neural machine translation with bi-directional GRU and attention mechanism on FPGAs using HLS," in *Proc. 24th Asia South Pacific Des. Autom. Conf.*, 2019, pp. 693–698.
[18] Y. Bengio *et al.*, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, no. 2, pp. 1137–1155, 2003.
[19] T. Mikolov *et al.*, "Recurrent neural network based language model," in *Proc. Annu. Conf. Int. Speech Commun. Assoc.*, 2010, pp. 1045–1048.
[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[21] J. Chung *et al.*, "Empirical evaluation of gated recurrent neural networks on sequence modeling," in *Proc. Annu. Conf. Neural Inf. Process. Syst. Workshop Deep Learn.*, 2014.
[22] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*.
[23] J. Gehring *et al.*, "A convolutional encoder model for neural machine translation," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 123–135.
[24] A. Vaswani *et al.*, "Attention is all you need," in *Proc. Conf. Neural Inf. Proces. Syst.*, 2017, pp. 5998–6008.
[25] C. Zhuge *et al.*, "Face recognition with hybrid efficient convolution algorithms on FPGAs," in *Proc. Great Lakes Symp. VLSI*, 2018, pp. 123–128.
[26] J. Wang *et al.*, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA," in *Proc. 28th Int. Conf. Field Programmable Logic Appl.*, 2018, pp. 163–169.
[27] S. Han *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 75–84.
[28] C. Gao *et al.*, "DeltaRNN: A power-efficient recurrent neural network accelerator," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2018, pp. 21–30.
[29] S. Cao *et al.*, "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2019, pp. 63–72.
[30] E. Nurvitadhi *et al.*, "Scalable low-latency persistent neural machine translation on CPU server with multiple FPGAs," in *Proc. Int. Conf. Field-Programmable Technol.*, 2019, pp. 307–310.
[31] C. Hao *et al.*, "FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge," in *Proc. 56th Annu. Des. Autom. Conf.*, 2019, pp. 1–6.
[32] X. Zhang *et al.*, "SkyNet: A hardware-efficient method for object detection and tracking on embedded systems," in *Proc. Conf. Mach. Learn. Syst.*, 2020, pp. 216–229.
[33] Y. Guan *et al.*, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *Proc. Asia South Pacific Des. Autom. Conf.*, 2017, pp. 629–634.
[34] P. Koehn, "Europarl: A parallel corpus for statistical machine translation," in *Proc. MT Summit*, 2005, pp. 79–86.
[35] K. Papineni *et al.*, "Bleu: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, 2002.



Qin Li received the BS and MS degrees in electrical and computer engineering from the University of Illinois at Urbana-Champaign (UIUC), in 2018 and 2020, respectively. She was a research assistant in CAD for Emerging System Group (ES-CAD) and she is currently working as a hardware engineer at Apple Inc.



Xiaofan Zhang (Student Member, IEEE) received the BS and MS degrees from the University of Electronic Science and Technology of China, in 2013 and 2016, respectively. He is working toward the PhD degree with the ECE Department, University of Illinois at Urbana-Champaign. His research interests include deep learning accelerator design, hardware-software co-design, and energy-efficient computing. He has received the IEEE/ACM William J. Mccalla ICCAD Best Paper Award, in 2018, IEEE/ACM Design Automation Conference System Design Contest double championships, in 2019, and Google PhD Fellowship, in 2020.



Jinjun Xiong (Senior Member, IEEE) received the PhD degree from the University of California, Los Angeles, CA, in 2006. He is a research staff member and program director for cognitive computing systems research with the IBM T.J. Watson Research Center. He cofounded and co-directs the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR). He is also an adjunct research professor with the ECE Department, UIUC. His research interests include AI, machine learning, and systems. His research was recognized with seven best paper awards and eight nominations for best paper awards at various international conferences.

Wen-Mei W. Hwu (Fellow, IEEE) joined NVIDIA, in February 2020 as senior distinguished research scientist, after spending 32 years at the UIUC, where he was a professor, Sanders-AMD Endowed Chair, acting department head and chief scientist of the Parallel Computing Institute. For his research contributions, he received the ACM SigArch Maurice Wilkes Award, ACM Grace Murray Hopper Award, IEEE Computer Society Charles Babbage Award, ISCA Influential Paper Award, MICRO Test-of-Time Award, IEEE Computer Society B. R. Rau Award, CGO Test-of-Time Award, and Distinguished Alumni Award in CS of the University of California, Berkeley. He has also won numerous best paper awards for major conferences. He is a fellow of ACM.

Deming Chen (Fellow, IEEE) received the BS degree in computer science from the University of Pittsburgh, in 1995, and the MS and PhD degrees in computer science from the University of California at Los Angeles, in 2001 and 2005, respectively. He is the Abel bliss endowed professor with the ECE Department, UIUC. His research interests include system-level and high-level synthesis, machine learning, computational genomics, GPU and reconfigurable computing, and hardware security. He has received the UIUC's Arnold O. Beckman Research Award, NSF CAREER Award, nine best paper awards, ACM SIGDA Outstanding New Faculty Award, and IBM Faculty Award. He an ACM distinguished speaker, and the editor-in-chief of the *ACM Transactions on Reconfigurable Technology and Systems* (TRETS).