EFFICIENT METHODS FOR MAPPING NEURAL MACHINE TRANSLATOR ON FPGAS

BY

QIN LI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Professor Deming Chen

ABSTRACT

Neural machine translation (NMT) is one of the most critical applications in natural language processing (NLP) with the main idea to convert text in one language to another language 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 thesis, we aim to exploit the capability of FPGAs for delivering highly efficient implementations for real-life NMT applications. In our work, 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 bidirectional GRU layer, attention mechanism, and beam search algorithm. 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 13.1x speedup with end-to-end performance of 22.0 GFLOPS without any accuracy degradation.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to express my special thanks to my adviser Professor Deming Chen and the ES-CAD research group at the University of Illinois at Urbana-Champaign.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 RELATED WORK 4
2.1 NMT 4
2.2 FPGA 5

CHAPTER 3 NMT MODEL 7
3.1 Overall Structure 7
3.2 Encoder 8
3.3 Decoder 9
3.4 Attention Mechanism 10
3.5 Beam Search 11

CHAPTER 4 PROFILING ANALYSIS 13
4.1 Computational Demand 13
4.2 Memory Overhead 14

CHAPTER 5 NMT HARDWARE DESIGN 16
5.1 NMT Overall Structure 16
5.2 Heterogeneous Decoders 17
5.3 Matrix-Vector Multiplication 19
5.4 Optimization Techniques 23
5.5 High-Level Synthesis 24

CHAPTER 6 SYSTEM-LEVEL ARCHITECTURE 26
6.1 HLS IP Integration 26
6.2 Off-chip Data Placement 27

CHAPTER 7 EXPERIMENTAL RESULTS 29
7.1 Prepare Work 29
7.2 Accuracy 29
7.3 Performance and Utilization 30
7.4 Comparison with Previous Work 31

CHAPTER 8 CONCLUSION 33

REFERENCES 34

CHAPTER 1

INTRODUCTION

Machine translation, converting source text in one language to text in another language automatically, 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]. 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 [2]. 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 [3, 4].

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 tasks [5]. 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 [6–10].

In our work, we mapped the inference of a classical encoder-decoder based NMT model proposed in [11] with total computation of 172 GFLOP to a highly optimized HLS IP and integrated 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 [12],

which is the first real-life NMT design on FPGAs using floating-point precision. 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 overheads of the targeted NMT 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.

- We propose **a heterogeneous attention-integrated decoder structure** to efficiently process the attention-based decoding process of recently developed NMT. Different decoding processes within the same NMT model can be treated specifically by the proposed decoders with different hardware configurations based on compute-to-communication (CTC) ratios.

- We introduce **highly optimized HLS IPs** as major building blocks for implementing NMT on FPGAs. To accommodate demanding NMT applications using limited hardware resources of the targeted single FPGA board, we propose HLS IPs with a variety of optimization techniques, such as partial on-chip weight storage, weight sharing, buffer sharing, optimized matrix-vector-multiplication (MVM) IPs, array partitioning, loop unrolling and pipelining, fine-tuning the degree of parallelism and resource sharing for the best performance result using HLS.

- We propose a **dedicated off-chip memory management** to relieve the tension of memory access from off-chip memory. We first rearrange the off-chip weights of each module following the computation order to enable the data burst transfer. Then, we fully utilize the available bus data width to activate and maximize the memory bandwidth occupation.

- To improve the inference speed, we leverage a **hybrid-precision model** on our board-level implementation on a Xilinx VCU118 FPGA and there is no accuracy loss. We quantized the NMT model based on both

performance and accuracy considerations and we increase the performance over the full floating-point version by 13.1x while maintaining the same accuracy.

The rest of this thesis is organized as follows. In Chapter 2, related work is introduced and the classical NMT model is described in Chapter 3. In Chapter 4, we present the profiling analysis on the targeted NMT model. Chapters 5 and 6 mainly focus on NMT HLS development and the way to integrate the NMT model to an FPGA board. In Chapter 7, we present experimental results, and in Chapter 8 we conclude this thesis.

CHAPTER 2

RELATED WORK

Many research projects focus on NMT algorithm design and FPGA acceleration. In this chapter, we briefly review previous works related to NMT model development and FPGA hardware implementations.

2.1 NMT

Models of machine translation are mostly sequence-to-sequence based, firstly encoding the input sentence and then decoding the encoded vectors to output text. Statistical machine translation applies 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 are applied to most of applications, researchers started to apply fully-connected layers and recurrent neural network (RNN) layers on machine translation models which outperform traditional SMT models [13, 14]. However, the accuracy is not stable when translating long sentences, so an attention mechanism, paying "attention" to related input vectors before generating each specific output word, was added to the model [11]. Even though standard RNN layers can connect the previous information to the current task, the previous information might be lost 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, was introduced to mitigate the issue, preserving portions of previous outputs in a memory cell [15]. By purely using six LSTM layers, the proposed work achieved better performance than previous works [1]. GRU (gate recurrent unit), another special RNN alternative, requires less computation than LSTM while still achieving the same accuracy [16]. To further improve the accuracy, Google proposed a large-scale NMT model, GNMT, with 16 LSTM layers and an attention

mechanism, and broke words into wordpieces [17]. Besides RNNs, Facebook used CNN (convolutional neural network) layers to encode sentences [18], and the model was outperformed by another NMT model, transformer, in which only various attention algorithms are used [19].

2.2 FPGA

Hardware acceleration is currently under significant exploration and development. FPGA has demonstrated its importance via various complex and effective applications. Lots of research projects related to DNN acceleration deployment on FPGA presented competitive energy efficiency and performance.

Matrix-vector multiplication processes are widely applied in the DNNs including fully connected (FC), RNN, and CNN layers. Even though there are all kinds of variations in NMT models, MVM is the main computation inside each of them. Sparse MVM accelerator engine that supports float MVM calculations with parallel compute engines was mapped to FPGA in [20]. However, the engine is still bounded by the memory bandwidth because the memory access is irregular and the loading process is on a single row of matrix. To mitigate the memory bandwidth limit, another sparse MVM engine was proposed with multi-row access and a novel sparse matrix encoding, which achieved a better result than the prior work [21]. Besides general MVM engines, FPGA-based accelerator engine design for CNN has also reached superior performance by applying loop tiling, unrolling, pipelining and data reuse under memory bandwidth and resource constraints [22]. ESE, an LSTM accelerator, provided high-performance and load-balancing acceleration with comprehensive sparsity and quantization exploration [23]. Other projects [24, 25] also presented alternative power-efficient and bank-balancing solutions on RNN accelerators. Another work proposed a dedicated CPU-FPGA system that is used for NMT acceleration while only MVM accelerator engines are mapped to FPGAs, and CPU handles all of the rest calculations [26]. Those previous FPGA 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 in order to run the entire DNN model, so that those designs did not adopt advantages of the energy efficiency on

FPGAs but brought another power-consuming source to CPU server.

There were also other works that deployed a complete network on an FPGA board. For example, lots of successful computer-vision related implementations have been deployed on FPGAs. An CNN accelerator design on embedded FPGA for ImageNet large-scale image classification using quantization techniques was proposed in [27]. In addition, a long-term recurrent convolutional network used for video content recognition was mapped to Virtex-7 FPGA board with high performance [5]. DNNbuilder, an automated tool for building high-performance DNN hardware accelerators for FPGAs, can directly map customized CNN and FC layers written in software to FPGAs with automatic optimization [28]. Compared with those DNNs which are CNN-dominated, our task is more challenging because most of the weights are used only once in MVM processes unlike convolutional calculations. Besides the computer vision field, an FPGA-based RNN implementation which can be used for NMT tasks with dedicated LSTM accelerator design was also implemented on a single FPGA [29]. However, the total number of operations for each previous network implemented on FPGAs is below 30 GOP, which is much smaller than our real-life NMT implementation with 172 GFLOP total computation demand.

In this thesis, we deploy a large-scale NMT model with mixed-precision on a single FPGA board after applying quantization techniques. To relieve the tension of memory access from off-chip memory, we provide dedicated design methodology of off-chip memory management and pipelined MVM kernel. Unlike other works which only focused on computational engine acceleration, we also provide guidance on how to generate a proper task partitioning strategy of a large-scale RNN model after profiling analysis in terms of computational demand and memory overhead under resource constraints. Finally, we adopt the benefits of the partition strategy by applying adaptive optimizations on the partitioned modules.

CHAPTER 3

NMT MODEL

Due to vanishing gradient issues, original RNNs (recurrent neural networks), which used to be widely applied in NMTs, have been replaced by GRU and LSTM layers [16]. In addition, adding attention mechanism preserves the accuracy of translating long sentences [11], and beam search algorithm limits the exponential number of results generated by the NMT model [1]. The NMT model we have adopted utilizes all of these features, and in this chapter, we demonstrate the detailed structure of this NMT model for FPGA implementation.

3.1 Overall Structure

The structure of the sequence-to-sequence based NMT model is presented in Figure 3.1. The model is equipped with bidirectional GRU layer for encoder, and attention mechanism and beam search algorithm 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 firstly 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, meaning that each of the dictionary contains 30000 words. If the word cannot be found in the dictionary, it will be marked as *UNK*.

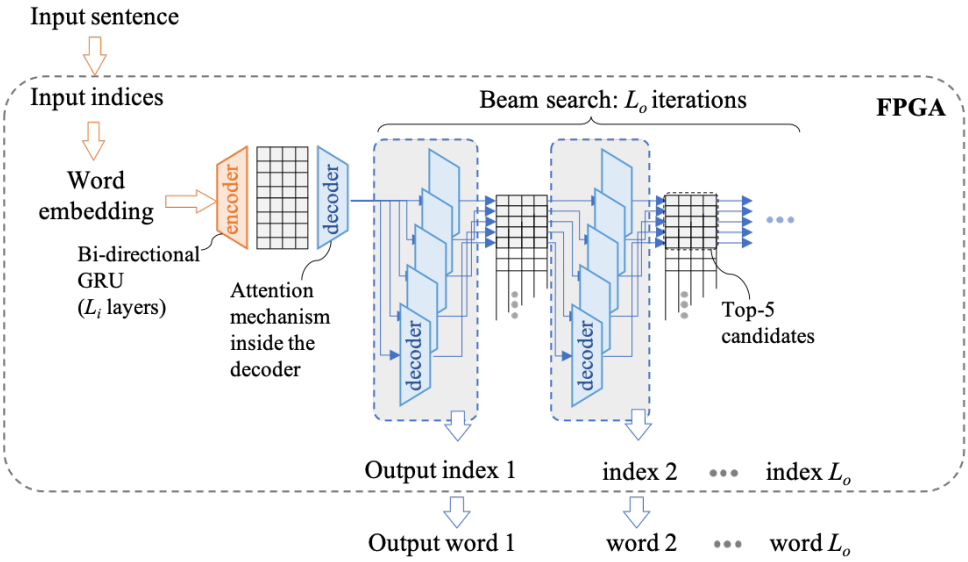

Figure 3.1: Overall structure of the targeted NMT model

3.2 Encoder

There are two inputs for the RNN layer: a vector denoting input word and another vector representing memory of this layer. We use GRU as the RNN layer in our NMT model, which requires less computation than LSTM while achieving similar accuracy [16]. Equation 3.1 is the updated gate representing the fraction the memory should be updated, while Equation 3.2, reset gate, represents the amount of memory that should be discarded. Equations 3.3 and 3.4 generate the current output according to the gate values. The current output vector is going to be the memory for the next iteration.

$$z^{(t)} = \sigma(W^{(z)}x^{(t)} + U^{(z)}h^{(t-1)}) \tag{3.1}$$

$$r^{(t)} = \sigma(W^{(r)}x^{(t)} + U^{(r)}h^{(t-1)}) \tag{3.2}$$

$$\tilde{h}^{(t)} = \tanh(U(r^{(t)} \circ h^{(t-1)}) + Wx^{(t)}) \tag{3.3}$$

$$h^{(t)} = (1 - z^{(t)})h^{(t-1)} + z^{(t)}\tilde{h}^{(t)} \tag{3.4}$$

Bidirectional RNN (with 2048 neurons) instead of a single RNN layer is

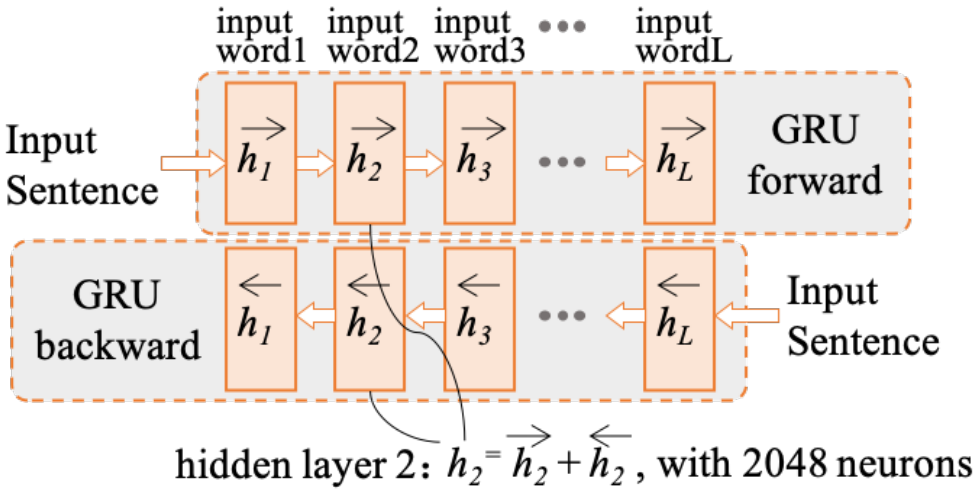

Figure 3.2: NMT encoder with bidirectional GRU layer

applied as the encoder, so that both words come before and after the current word can be linked. As shown in Figure 3.2, 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 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 Figure 3.3, the previous output word embedding vector and encoder output are the inputs for each iteration.

Firstly, a GRU layer links all of the previous output words, and both

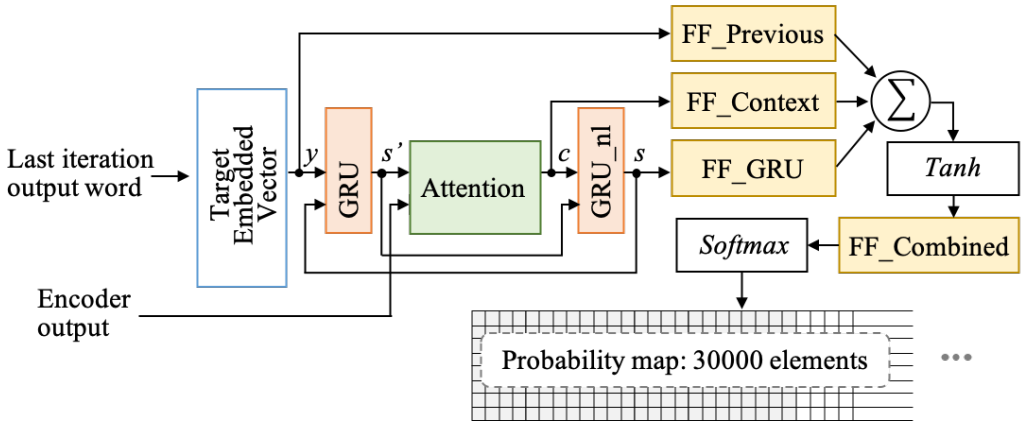

Figure 3.3: Detailed decoder structure

GRU output, s, and encoded vectors, h_i, are sent to attention mechanism. Considering that RNN layers have two outputs, memory and the input vector, 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 1024 neurons. The output of GRU_nl, 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 each with 512 neurons, $FF\_Previous$, $FF\_Context$, FF_GRU, respectively. Summations of FC outputs are sent to another FC layer with softmax function, generating a probability map, a vector with 30000 elements each representing a probability of a specific word in the target dictionary. The beam search algorithm determines which words to choose as the next outputs.

3.4 Attention Mechanism

$$e_{ij} = v^\top \tanh(U_a s'_j + W_a h_i) \tag{3.5}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_i^{L_i} \exp(e_{ij})} \tag{3.6}$$

$$c_j = \sum_{i}^{L_i} \alpha_{ij} h_i \tag{3.7}$$

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 for each input word when generating the current output word, 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 3.5, 3.6 and 3.7 in the NMT model [11]. In each decoder 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 with the same length as the target dictionary 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 simplify the task, while at the same time approaching the optimal sentence as much as possible, we use the beam search algorithm. The algorithm 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 Figure 3.4, and among the 5×30000 scores generated, the next top five

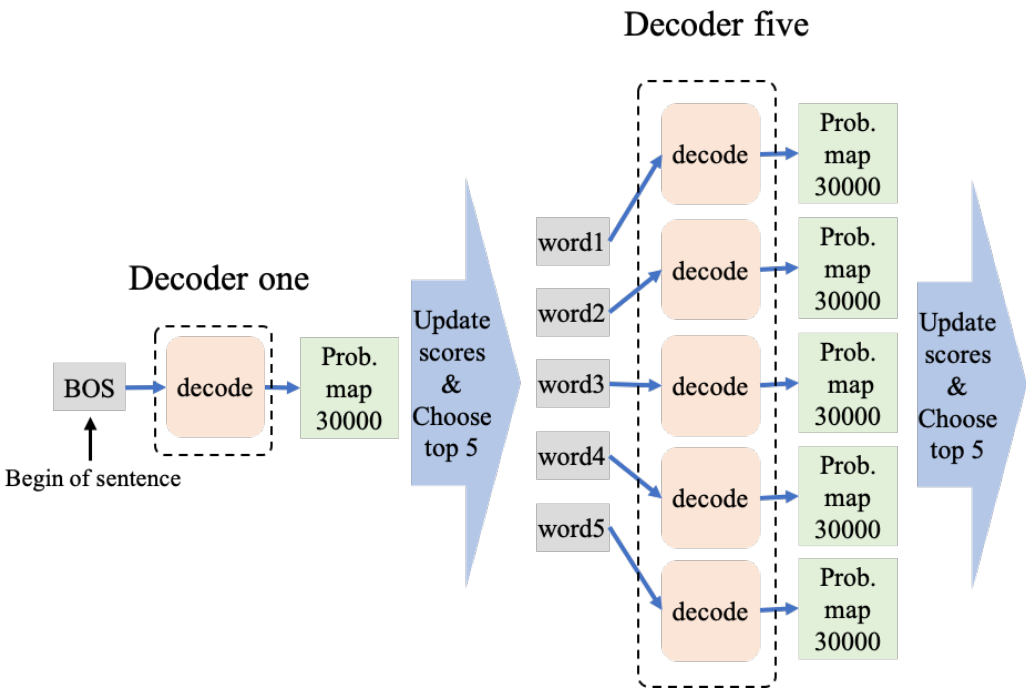

Figure 3.4: Beam search with two kinds of decoding iterations

potential answers are preserved. The translation process terminates when all five tracks reach *eos*, where the sentence with the highest score is selected.

CHAPTER 4

PROFILING ANALYSIS

Unlike instruction-based CPUs and GPUs, FPGA takes the advantage of customizability. Instead of compiling the DNN model to compatible machine code, redundantly fetching instructions and using the same compute engine, DNN deployment on FPGAs allows direct computing and data transferring on hardware and the parallelism of each layer is adjustable. However, each FPGA board 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 chapter, we present our profiling results and analyze the computational demand and memory overhead of the NMT model.

4.1 Computational Demand

Equipped with GRU, FF, attention mechanism layers and beam search algorithm, the targeted NMT model requires both computation of linear matrix-vector multiplication and non-linear activation functions including softmax, hyperbolic tangent, and sigmoid. The log function is also needed for calculating scores of the potential output sentences. Among those calculations, linear matrix-vector multiplication dominates the total computation.

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 Figure 4.1. 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% 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

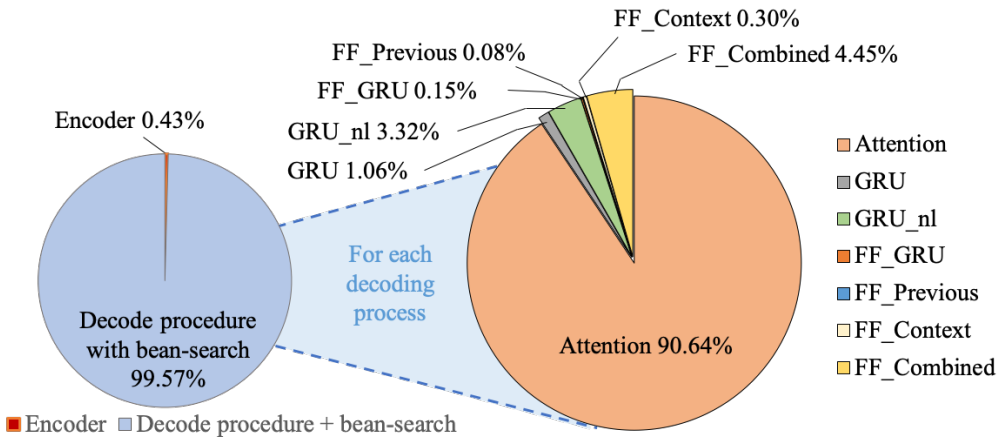

Figure 4.1: Operation distribution in the targeted NMT (left) and computational demand breakdown in a single decoding process (right)

decoding process, attention mechanism has the most computation (90.64% of the decoding process) due to calculation of relativity values between 50 encoded vectors and the current output word. 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.

4.2 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 compu-

tation 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.

CHAPTER 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 chapter, 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

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 requires LUT (lookup table), FF (flip-flop) and DSP (a computational unit) resources. BRAM and URAM are the main on-chip memory storage and DRAM is storage off-chip. Overall accelerator design is shown in Figure 5.1.

In NMT logic, the translator consists of an encoder and an decoder. According to the profiling results in Figure 4.1, 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. For the encoder and Decoder one, only one MVM kernel module

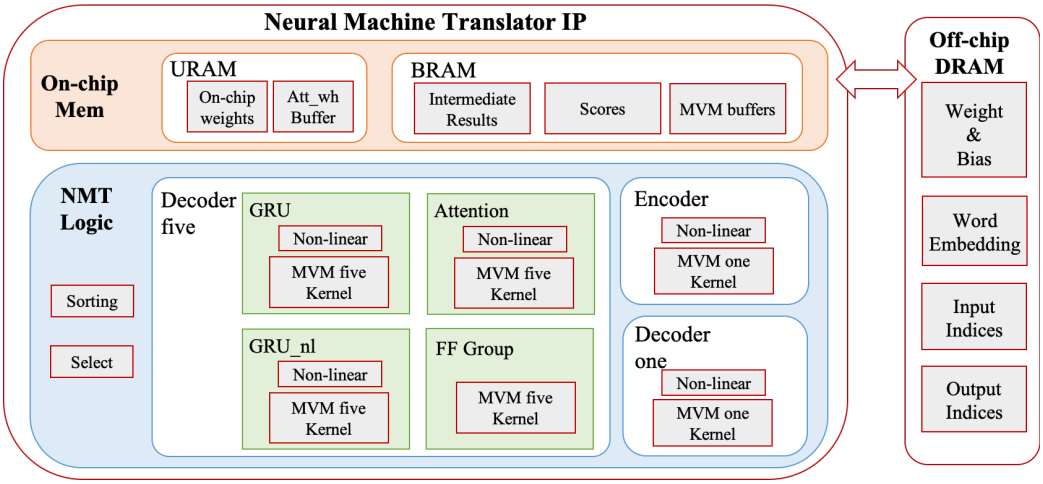

Figure 5.1: Accelerator-level structure of NMT

is shared by all of MVM processes, while multiple MVM kernels are instantiated in the most computationally demanded Decoder five module. 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 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 Figure 3.4, two kinds of decoding processes are executed during translation. We separate them into different modules, Decoder five and Decoder one, for two reasons. Firstly, 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 vs Decoder One

Decoder five is responsible for most of calculations in NMT. Five decoding processes run concurrently inside Decoder five 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 mechanism module. The optimization technique greatly reduces redundant calculations, so that the calculation is firstly done and results are stored by Decoder one for re-usage in Decoder five.

Among all of the calculations, attention mechanism accounts for 90% of them according to our profiling results shown in Figure 4.1. The large amount of calculation is used for computing the energy state of each of L_i encoded vectors, e_{ij} in equation (3.5), 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 Figure 5.2. In our design, $U_a s'_j$ is firstly calculated and stored temporarily for both MVM calculations in Decoder one and Decoder five. $W_a h_i$ is only calculated in Decoder one.

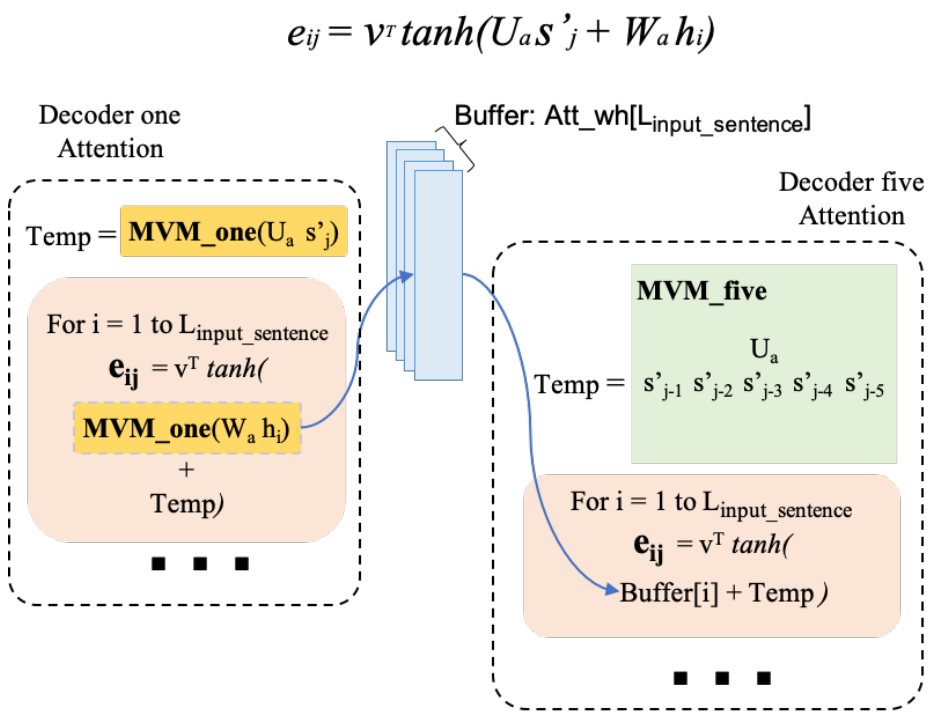

Figure 5.2: Optimized flow for attention mechanism

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 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.

Table 5.1: 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

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 5.1, 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 Figure 5.3, 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 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 Figure 5.3, 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, firstly 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 vs 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.

We used CE64 in our MVM design, so each CE receives 64 pairs of half

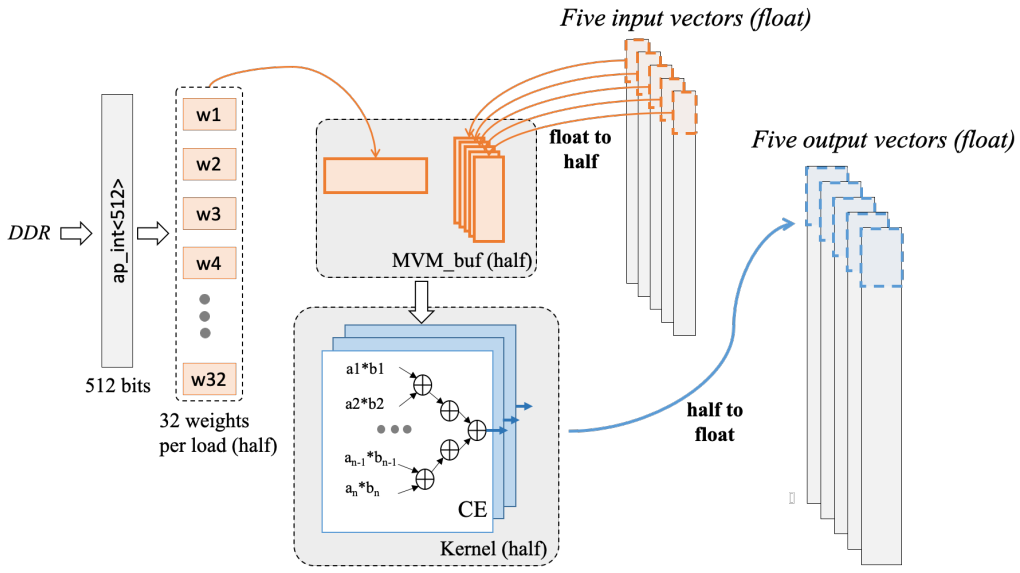

Figure 5.3: MVM five kernel design with mixed precision in Decoder five

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. However, 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.

5.3.4 Float MVM vs 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 Figure 5.4. 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

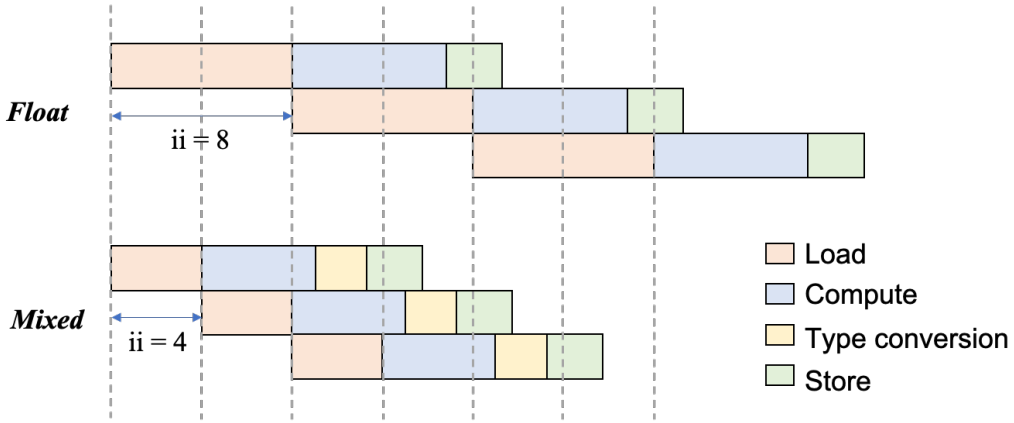

Figure 5.4: Timing diagram comparison between float MVM and mixed-precision MVM

loading speed and thus under memory bounded situation has 2x speedup.

5.4 Optimization Techniques

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 Figure 5.5.

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

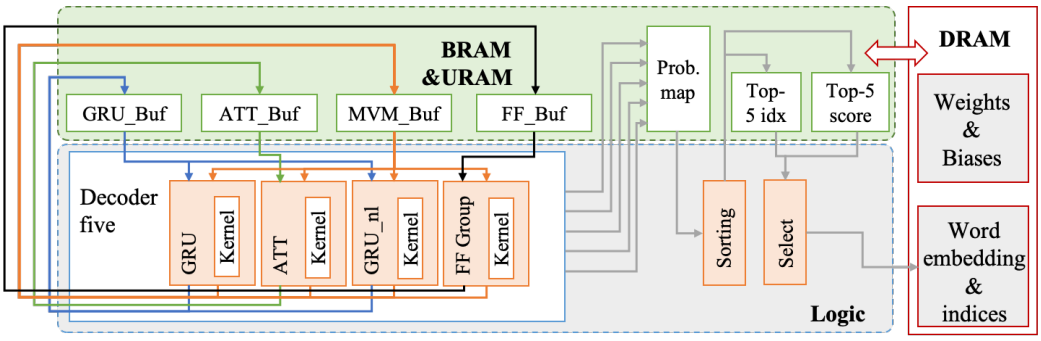

Figure 5.5: Detailed structure of Decoder five and other related logic

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 Figure 5.5, 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.

5.5 High-Level Synthesis

High-level synthesis, using behavioral languages (C/C++) to design hardware, can significantly improve the productivity of FPGA design and provide

opportunities for efficient design space exploration, especially when we target such a large-scale NMT model with total computation of 172 GFLOP. We realize our hardware design methodology using HLS.

In order to design a translator using HLS, we need to firstly 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. A port is added to modules to load data from off-chip memory.

To realize our optimization techniques under resource constraints, specific pragmas are added to notify the HLS compiler for special optimizations such as loop unrolling and pipelining, global buffers are created for sharing memory resources between modules, and computational IPs are reused inside modules by using the same C function interface to reduce DSP and LUT usage while maintaining the performance.

CHAPTER 6

SYSTEM-LEVEL ARCHITECTURE

To have a real board-level implementation of NMT, we need to integrate the HLS IP into a system-level architecture design and generate a bitstream for the targeted FPGA board. After mapping the bitstream to the device, we use a program running on host CPU to control the HLS IP on the FPGA board. In this chapter, we demonstrate our system-level design methodology.

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.

The overall architecture design is shown in Figure 6.1. We applied a memory-map based mechanism to complete the handshake between the FPGA board (translator side) and the host CPU (host side). The architecture includes off-chip DDR memory, DMA, HLS translator 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.

The detailed flow of each translation process is as follows. The host CPU initially sends the input binaries into DDR memory using PCIe port. After the data transfer completes, the CPU activates the translator by setting *ap_start* signal to one. The host CPU continuously checks *ap_idle* signal through PCIe port to see if the translator completes its work. Once the translator finishes, it puts the outputs back to DDR memory and toggles

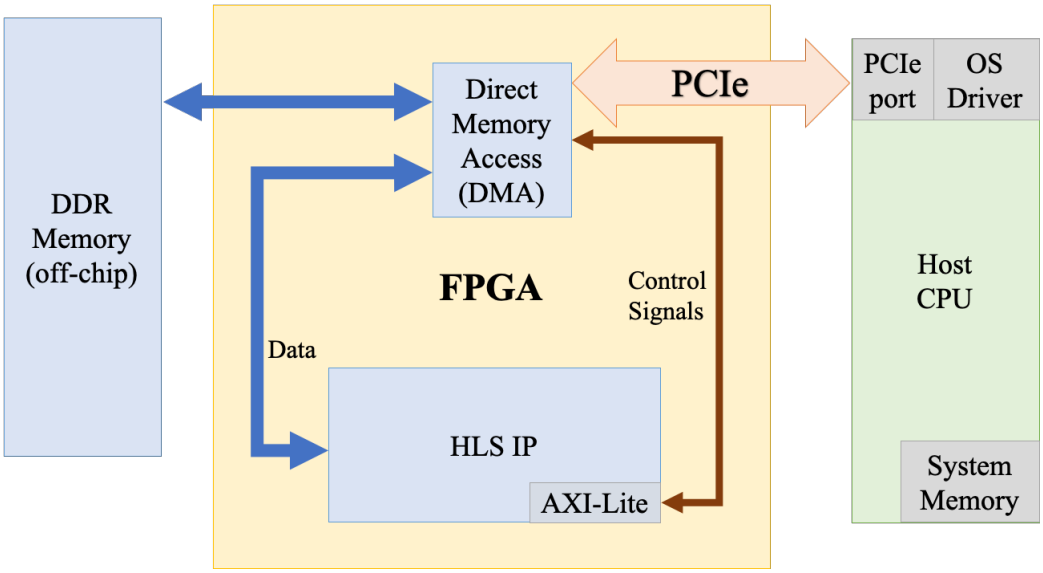

Figure 6.1: System-level architecture for the NMT model

the *ap_idle* signal. Then the host CPU reads a specific region of the off-chip memory to obtain the results.

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 Figure 6.2, 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 re-arranged 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 applying 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

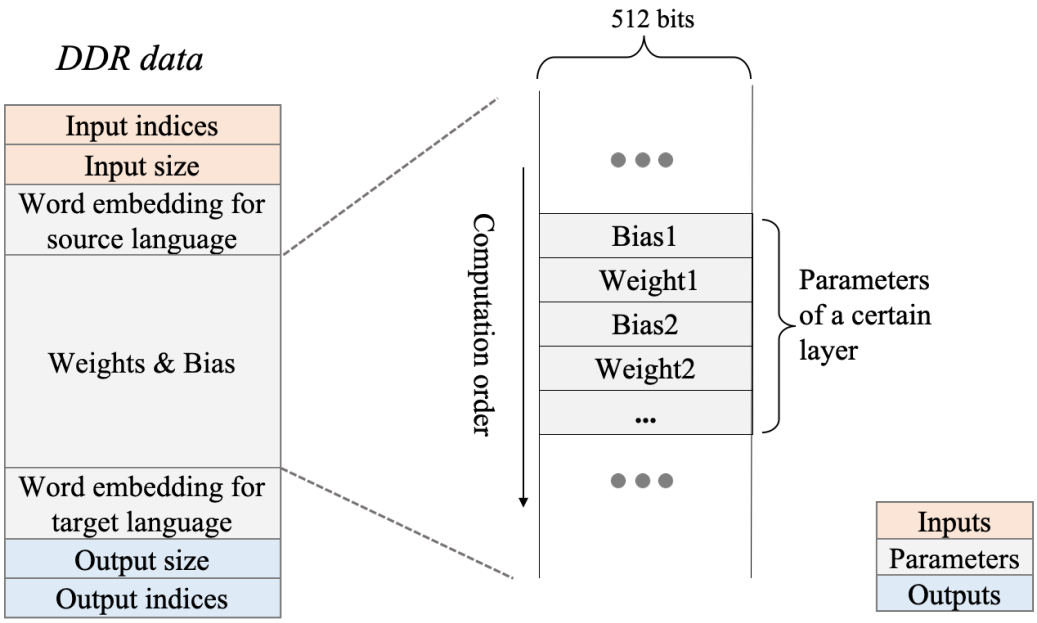

Figure 6.2: Data placement on off-chip DRAM

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.

CHAPTER 7

EXPERIMENTAL RESULTS

In this chapter, we map the classic NMT model to Xilinx Virtex UltraScale+ VCU118 FPGA development board and demonstrate the experimental results of our work in terms of accuracy, performance, and resource utilization.

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 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 [30].

7.2 Accuracy

In Pytorch, both half and float weights are trained for the NMT model. As shown in Table 7.1, Pytorch Float32 denotes the floating-point implementation in Pytorch while Pytorch Half16 represents the same model with half-precision weights. We calculate BLEU scores (a standard set of criteria to evaluate the quality of text [31]) for the two models as well as our work

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

Table 7.1: Accuracy comparison on WMT English → French (newstest2014)

Model	Pytorch Float32	Pytorch Half16	our work
BLEU	22.3	21.9	22.6

running on WMT14 (Conference on Machine Translation) English-French test dataset. As shown in Table 7.1, 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.

7.3 Performance and Utilization

Multiple NMT designs are compared in terms of performance and resource utilization in Table 7.2. The resource utilization information for each design is represented as percentage values of the Ultrascale+ FPGA chip resources, and 50-word Translation in the table means the time needed for translating 50-word English sentence to 50-word French sentence. We have four models, **Float**, **Orig**, **Onchip**, and **Optimized**, where **Float** represents the original float NMT model implemented in [12], **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.

7.3.1 Float vs 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 7.2, 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

Table 7.2: Utilization and performance comparison between different NMT implementation

Model	Float	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

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% 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 with Previous Work

In the previous NMT work [12], 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. In addition,

Table 7.3: Comparison with previous work

Reference	[29]	Float NMT [12]	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

the previous design used too much resource and the router could not resolve congestion issue, so we lowered the level of parallelism for the design in order to map the design to the board. The corrected result along with result of another ASPDAC17 work [29] is presented in Table 7.3. We calculated the end-to-end performance by the information provided in the ASPDAC17 paper (problem set size: 2.76 mega-operations divide by overall latency: 0.39 second) [29]. As shown in Table 7.3, our work has much better performance than previous works with our optimized implementation. Compared to the previous float NMT design [12], we achieve 13.1x speedup after all of our optimization techniques with end-to-end performance of 22.0 GFLOPS.

CHAPTER 8

CONCLUSION

In our work, we map a 172 GFLOPS 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 design achieves 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 play with small-scale DNN models, we implement 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 are able to fine-tune our NMT design with much less effort than using traditional HDL to achieve optimal performance under FPGA resource constraints.

For future work, state-of-the-art NMT models such as Transformers can be targets for FPGA deployment. Cloud FPGA clusters with much more resources are also suitable hardware to hold large NMT models. For further acceleration, pruning and fixed-point quantization techniques are potential choices.

REFERENCES

[1] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems*, 2014, pp. 3104–3112.

[2] M. X. Chen, O. Firat, A. Bapna, M. Johnson, W. Macherey, G. Foster, L. Jones, N. Parmar, M. Schuster, Z. Chen et al., "The best of both worlds: Combining recent advances in neural machine translation," *arXiv preprint arXiv:1804.09849*, 2018.

[3] J. Quinn and M. Ballesteros, "Pieces of eight: 8-bit neural machine translation," *arXiv preprint arXiv:1804.05038*, 2018.

[4] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik, "Neural network distiller: A Python package for DNN compression research," *arXiv preprint arXiv:1910.12232*, 2019.

[5] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen, "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–4.

[6] Xilinx, "Vivado high-level synthesis." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[7] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "xpilot: A platform-based behavioral synthesis system," *SRC TechCon*, vol. 5, 2005.

[8] D. Chen, J. Cong, Y. Fan, and L. Wan, "LOPASS: A low-power architectural synthesis system for FPGAs with interconnect estimation and optimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 4, pp. 564–577, 2009.

[9] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2011, pp. 33–36.

[10] X. Liu, Y. Chen, T. Nguyen, S. Gurumani, K. Rupnow, and D. Chen, "High level synthesis of complex applications: An H. 264 video decoder," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 224–233.

[11] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[12] Q. Li, X. Zhang, J. Xiong, W.-m. Hwu, and D. Chen, "Implementing neural machine translation with bi-directional GRU and attention mechanism on FPGAs using HLS," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019, pp. 693–698.

[13] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of Machine Learning Research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[14] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[16] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[17] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey et al., "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[18] J. Gehring, M. Auli, D. Grangier, and Y. N. Dauphin, "A convolutional encoder model for neural machine translation," *arXiv preprint arXiv:1611.02344*, 2016.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[20] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, 2005, pp. 63–74.

[21] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2014, pp. 36–43.

[22] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[23] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang et al., "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 75–84.

[24] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "DeltaRNN: A power-efficient recurrent neural network accelerator," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 21–30.

[25] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 63–72.

[26] E. Nurvitadhi, A. Boutros, P. Budhkar, A. Jafari, D. Kwon, D. Sheffield, A. Prabhakaran, K. Gururaj, P. Appana, and M. Naik, "Scalable low-latency persistent neural machine translation on CPU server with multiple FPGAs," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 307–310.

[27] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song et al., "Going deeper with embedded FPGA platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[28] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[29] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 629–634.

[30] P. Koehn, "Europarl: A parallel corpus for statistical machine translation," in *MT Summit*, vol. 5. Citeseer, 2005, pp. 79–86.

[31] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.