Heterogeneous Model to Heterogeneous System Mapping with Computation and Communication Awareness

by

Xinyi Zhang

B.E., Southwest Jiaotong University, 2014

M.S., University of New Mexico, 2016

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2022

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Xinyi Zhang

It was defended on

February 28 2022

and approved by

Jingtong Hu, Ph.D., Associate Professor

Department of Electrical and Computer Engineering

Alex Jones, Ph.D., Professor

Department of Electrical and Computer Engineering

Samuel Dickerson, Ph.D., Associate Professor

Department of Electrical and Computer Engineering

In Hee Lee, Ph.D., Assistant Professor

Department of Electrical and Computer Engineering

Youtao Zhang, Ph.D., Professor

Department of Computer Science

Callie Hao, Ph.D., Assistant Professor

Georgia Institute of Technology

Dissertation Director: Jingtong Hu, Ph.D., Associate Professor

Department of Electrical and Computer Engineering

Heterogeneous Model to Heterogeneous System Mapping with Computation and Communication Awareness

Xinyi Zhang, PhD

University of Pittsburgh, 2022

While machine learning (ML) has been widely used in real-life applications, the complex nature of real-world problems calls for heterogeneity in both machine learning models and hardware systems. For the algorithm, the heterogeneity in ML models comes from the multi-sensor perceiving and multi-task learning, i.e., multi-modality multi-task (MMMT) models, resulting in diverse Deep Neural Networks (DNNs) with associated DNN layers. For the system, as the diverse DNN layers largely increase the heterogeneity of computing and dataflow patterns, heterogeneous computing becomes a promising solution to address the computation efficiency. it becomes prevailing to integrate dedicated acceleration components such as CPU, GPU, ASIC, and FPGA accelerators into one system to improve overall efficiency. It thus introduces a new problem, heterogeneous model to heterogeneous system mapping (H2H), in which both computation and communication efficiency need to be considered.

This dissertation proposes three aspects to enable an efficient heterogeneous model to heterogeneous system mapping. First, a Convolution accelerator design exploration based on FPGA is proposed to address the efficiency in CNN models. Second, the accelerator architecture exploration for LSTM and Transformer are proposed based on FPGA to address the computing efficiency for LSTM and Transformer based models. Third, aiming at system-level formulation, simulation, and optimization, a computation and communication aware heterogeneous model to heterogeneous system mapping algorithm with its associated system-level simulator is proposed. In solving the H2H system mapping, we take the multi-FPGA system as the multi-accelerator platform due to its flexibility in accelerator architecture exploration and fast prototyping procedure.

Table of Contents

Preface . . . xii

1.0 Introduction . . . 1

1.1 Challenges in Deploying Multi-modal Multi-task Model to Multi-accelerator 3

1.2 Research Contributions . . . 4

1.3 Dissertation Organization . . . 5

2.0 Background . . . 6

2.1 Heterogeneous DNN Model . . . 6

2.2 Convolution in Computer Vision Domain . . . 9

2.3 LSTM and Transformer in Natural Language Processing Domain . . . 10

2.3.1 Long Short-term Memory . . . 11

2.3.2 Transformer . . . 12

2.4 Heterogeneous System . . . 14

3.0 Accelerator Optimization for Convolution Neural Networks (CNN) . . . 18

3.1 Background . . . 18

3.2 Motivation . . . 21

3.3 Convolution Accelerator and Analytical Model . . . 22

3.4 Experiment . . . 29

3.4.1 Experiment Setup . . . 29

3.4.2 Performance Evaluation . . . 29

3.5 Summary . . . 33

4.0 Software/Hardware Co-design for LSTM and Transformer Network . . . 35

4.1 Accelerator Design for Long Short-term Memory (LSTM) . . . 35

4.1.1 Background . . . 35

4.1.2 Motivation and Contribution . . . 37

4.1.3 LSTM Accelerator and Analytical Model . . . 38

4.1.3.1 Unifying the Computing Patterns . . . 38

4.1.3.2 Accelerator based on the Unified Pattern 41
4.1.3.3 Analytical Model 43
4.1.4 Experiment 46
4.1.4.1 Experiment Setup 46
4.1.4.2 Performance Evaluation 46
4.1.4.3 Analytical Model Accuracy Analysis 49
4.2 Software/Hardware Co-design for Transformer Network 50
4.2.1 Background 50
4.2.2 Motivation and Contribution 52
4.2.3 Algorithm Optimization 53
4.2.3.1 Weight Significance Analysis before Model Compression 53
4.2.3.2 Pruning Strategy 56
4.2.4 Hardware Optimization 59
4.2.4.1 Unified Computing Pattern in Sparse Transformer 59
4.2.4.2 Accelerator Design 60
4.2.4.3 Accelerator Analytical Model 67
4.2.5 Experiment 68
4.2.5.1 Experiment Setup 68
4.2.5.2 Model Compression Performance 69
4.2.5.3 Accelerator Performance 69
4.3 Summary 75
5.0 Multi-modal Multi-task Model to Multi-accelerator Mapping 76
5.1 Background 76
5.2 Motivation 80
5.3 Methodology 81
5.3.1 System Formulation 81
5.3.2 Mapping Algorithm 84
5.3.3 Experiment 88
5.3.3.1 Experiment Setup 88
5.3.3.2 Mapping performance 88

5.3.3.3 Performance comparison and mapping uncertainty discussion . 90
6.0 Conclusion and future work 98
Bibliography 99

List of Tables

1 Experimental results of 2-FPGA system with comparisons to the existing designs in accelerating AlexNet 5 Convolution layers 31
2 Comparison results on 2-ZCU102 32
3 Convolution accelerator performance model accuracy analysis. 34
4 Performance comparison for designs on 7Z020 47
5 Performance details of the proposed design 48
6 Resource comparison for designs on 7Z020 48
7 Power comparison for designs on 7Z020 48
8 Accelerators hyperparameters for 7Z020 50
9 LSTM accelerator performance model accuracy analysis. 50
10 Transformer parameters 69
11 Performance of compression and quantization 70
12 Processing element resource breakdown 70
13 Accelerator buffer allocation 71
14 Accelerator performance on ZCU102 71
15 Transformer accelerator performance model accuracy analysis. 71
16 End-to-end accelerator performance on real Transformer 74
17 System performance modeling parameters 82
18 Heterogeneous models 88
19 State-of-the-art FPGA DNN accelerators 89
20 Latency reduction breakdown comparing with the second step (baseline). 92
21 Latency reduction breakdown comparing with the second step (baseline). 92
22 The mapping performance and searching time performance comparison. 93
23 The mapping performance and searching time performance comparison. 93
24 The mapping uncertainty analysis. 93

List of Figures

1 Multi-modal multi-task model overview. 2
2 VLocNet++:semantic visual localization. 7
3 MMMT model for Camera-Radar data analysis. 7
4 MMMT model for Emotion Recognition. 8
5 A convolution layer and its computing Pseudo code 10
6 Fully connected layer breakdown. 10
7 LSTM architecture. 11
8 Computations in self-attention. 13
9 Multi-head self-attention. 13
10 Transformer encoder and decoder. 14
11 The state-of-the-art multi-accelerator chip 17
12 Multi-FPGA system. 17
13 AlexNet architecture. 19
14 Convolution operation. 20
15 ReLu activation. 20
16 MaxPool. 20
17 NVDLA and Shi-Dinanao Style Accelerator. 22
18 Convolution layer tiling. 22
19 Convolution accelerator architecture. 24
20 Convolution accelerator computing kernel. 24
21 Optimized Convolution nested loops. 24
22 Convolution accelerator performance bottleneck. 25
23 The transmission and computation workload sharing 28
24 Two-FPGA data sharing examples. 28
25 Two-FPGA system. 29
26 Power measurement of on-board executions. 30

27 Comparisons of predictable models and on-board executions on latency: employing different designs on single-FPGA and two-FPGA systems. 34
28 The matrix-vector multiplication for $W_x x_t$. 36
29 The matrix-vector multiplication for $W_h h_{t-1}$. 36
30 The element-wise addition and multiplication. 36
31 The computing kernels of LSTM accelerator. 37
32 The data dependency of computing kernels in the LSTM accelerator. 37
33 Applying fundamental patterns to $w_x x_t$. 40
34 Applying fundamental patterns to $w_h h_{t-1}$. 40
35 Applying fundamental patterns to a gate. 40
36 Applying fundamental patterns to four gates. 41
37 The unified-kernel LSTM accelerator architecture. 43
38 Physical buffer size and weight size. 44
39 Power measurement of accelerators on 7Z020. 49
40 Related works. 53
41 LayerNorm insertion of the encoder. 55
42 LayerNorm scaling factor visualization. 56
43 Sparse self-attention computations. 59
44 Sparse FFN computations. 60
45 Unified computing pattern. 61
46 Loop iteration of the unified computing pattern. 61
47 INT8 multiplication encoding. 61
48 (a) The PE1 architecture and PE1 mapping to the multiplication in the unified computing pattern. (b) The PE2 architecture and PE2 mapping to the addition in the unified computing pattern. 62
49 (a) Computing core hierarchy. (b) Accelerator architecture overview. 63
50 The data flow of the computing core. 63
51 The accelerator running schedule. 67
52 Multi-modal multi-task model. 76
53 The existing multi-modal multi-task models. 78

54 Convolution and Fully-connection layer 79
55 An example of communication-prioritized mapping and communication-aware mapping. The later slightly sacrifices the computation efficiency but reduces the overall system latency by avoiding expensive data movement. 80
56 H2H mapping algorithm visualization. It includes 4 major steps: (1) computation-prioritized mapping; (2) weight locality optimization; (3) activation transfer optimization; (4) data locality aware remapping. 83
57 The latency and energy performance comparison. 90
58 Communication and computation ratio. 91
59 The visualization of system latency non-linear increment 93

Preface

First of all, I want to thank my Ph.D. advisor, Prof. Jingtong Hu, for the continuous support of my Ph.D. study and research. I appreciate all his ideas, patience, enthusiasm, and perseverance in academic pursuits. I appreciate his constructive advice on my career plan and his encouragement when I started my Ph.D. study.

I reserve my sincere gratitude to Prof. Alex Jones, Prof. Samuel Dickerson, Prof. Inhee Lee, Prof. Youtao Zhang, and Prof. Callie Hao, for serving on my Ph.D. committee and providing insightful advice and instructions to my research and dissertation.

I have been fortunate to work with my group comrades, Mimi Xie, Chen Pan, Yawen Wu, Zhenge Jia, Zhepeng Wang, and Yue Tang, who are always generous with their time and knowledge. They have made my life vivid and beautiful during the past years.

Last but not least, I would also like to thank my parents and grandparents for their continuous support and unconditional love to me. I dedicate this dissertation to them!

1.0 Introduction

As DNNs are applied in more and more complicated applications, both the models and hardware acceleration systems call for **heterogeneity** [1, 2] to address rising challenges. First, the ML algorithms are evolving from handling single-modality single-task to multi-modality multi-task (MMMT) [1]. For instance, in the recommendation system, visual and textual data are jointly learned in a multi-modality fashion for better prediction performance [3], and in AR/VR, image, gesture, and speech are jointly learned for better rendering quality [4]. Such changes result in increasingly complicated DNN models with larger size and complex inner model dependency. Second, recent advanced systems are introducing great heterogeneity by integrating different acceleration components with diverse capabilities to achieve both low latency and high energy efficiency. Microsoft's Brainwave [5] adopts a hybrid CPU-FPGA cloud architecture to accommodate different computation tasks. AWS [6] integrated multi-FPGA in their cloud to enable heterogeneous computing to improve computing efficiency. Other CPUs, GPUs, and FPGAs based heterogeneous multi-accelerator designs are seen in [5, 6, 7, 8, 9, 10, 11].

Multi-modality multi-task models usually consist tasks such as image classification, object detection, gesture recognition, machine translation, etc., in both CV and NLP domains. This results in diverse DNN layers in the MMMT model, such as Convolution, Fully-connection (FC), LSTM, Transformer, Pooling, SoftMax, Normalization, etc. Different types of DNN layers are specialized in the computation and memory pattern requirement. Among the DNN layers, Convolution, Fully-connection, LSTM, and Transformer are widely used as the main modules of multi-modality multi-task models. Convolution is primarily sliding window based Multiply Accumulate (MAC). Fully-connection is primarily matrix multiplication. LSTM is primarily matrix multiplication and addition. The Transformer is primarily matrix multiplication, addition, division, and array transposing. To enhance the performance, the MMMT models also widely adopt inter-block connection (inner model dependency) between modality net or task net in order to share data between sub-models. An example of MMMT for VR is shown in Figure 1. The captured data in visual, audi-

tory, thermal, and olfactory are processed by corresponding modality nets and the task nets. Among the sub-nets, inner model dependencies are applied for better information relation. The complex inter-block connections in the model lead to more DNN layer dependency and data movement in MMMT than the general uni-modal uni-task DNN. The computing burden and model dependency of the multi-modality multi-task model increase with the growing complexity of the application.

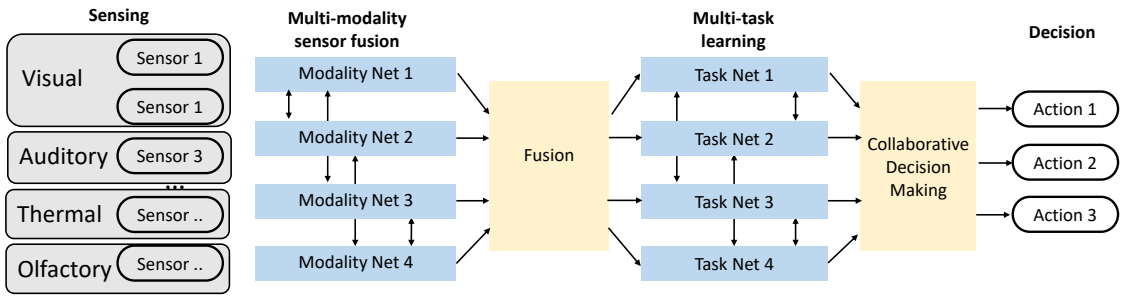


Figure 1: Multi-modal multi-task model overview.

To address the computing challenges of DNN models, researchers have explored different accelerator architectures towards different DNN layers. [12, 13, 14, 15, 16, 17] proposed CNN and FC accelerators with the optimizations such as loop-tiling, loop-reordering, line-buffer, etc,. [18, 19, 20] proposed LSTM accelerators with the optimizations of the data flow in the model. [21, 22] proposed Transformer accelerators with the optimization of the computation parallelism within the self-attention mechanism.

The computing efficiency of the multi-accelerator system in processing multi-DNN is explored by [5, 23, 24]. However, these works explored the efficiency of processing several standalone DNN models via a multi-accelerator system simultaneously. [5] improves the single accelerator computing efficiency in the multi-accelerator system by augmenting the data flow but system-level cross-accelerator communication is not discussed. [23] maps DNN layers to different accelerators to fully utilize DSP and block-RAM resources in the individual accelerator. [24] proposes the computation prioritized mapping, pairing the DNN layers with preferable accelerator considering both computing pattern and data flow in the accelerator. Despite the diversity of the computing workload from DNN and multi-DNN are addressed by multi-accelerator design, the inter-block data sharing in the emerging multi-modal multi-task

DNN models cause non-negligible transmission overhead. Deploying the multi-modal multi-task DNN models to the multi-accelerator system still face challenges from high computation and communication overhead.

1.1 Challenges in Deploying Multi-modal Multi-task Model to Multi-accelerator

Deploying a multi-modal multi-task DNN model on a multi-accelerator system needs an in-depth understanding of accelerators for different DNNs and the architecture of the multi-accelerator system. This dissertation considers three main challenges in deploying a multi-modal multi-task DNN model to a multi-accelerator system.

First, the computing efficiency of DNN models in the Computer Vision domain should be fully addressed. The DNN models in the CV domain are primarily CNN-based models, which usually face high data computing and transmission burden caused by convolution. However, hardware accelerators such as ASICs and FPGAs as the on-chip storage scarce platforms, can hardly accommodate a CNN model or even a single Convolution layer on the chip. The optimization of Convolution kernel design, Convolution workload partition, and the associated accelerator performance model is needed.

Second, the computing efficiency of DNN models in Natural Language Processing domain on the accelerator should also be fully addressed. The DNN models in the NLP domain are primarily LSTM or Transformer based models. In the contrast to CNN models, LSTM and Transformer are generally sophisticated in its algorithm design while demanding less computation workload. The optimization of the computation and the data flow of the LSTM and Transformer in the accelerator is needed.

Third, the mapping of multi-modal multi-task DNN model to multi-accelerator that considers both computation and communication efficiency is still missing. The mapping needs the formulation to depict layer dependency in multi-modal multi-task DNN model, multi-accelerator system architecture, accelerator performance, and the execution dependency of the accelerators. Therefore, a system-level formulation, modelling, and optimization are vi-

tally needed in the mapping of multi-modal multi-task DNN model to the multi-accelerator. In this dissertation, these challenges are addressed. The goal is to enable reliable, fast, and robust mapping of multi-modal multi-task DNN model to multi-accelerator system.

1.2 Research Contributions

Research contributions for this dissertation can be concluded as:

- To address the computing efficiency of DNN models in the Computer Vision domain, a Convolution accelerator design on the FPGA with a performance model is proposed. Specifically, the proposed Convolution accelerator makes the following contributions:
  - A computing kernel with the associated hardware accelerator is designed that achieves optimal pipeline performance for Convolution.
  - A Convolution tiling strategy is proposed to partition the oversized Convolution layer to tiles that can be iteratively processed by the accelerator.
  - A performance model is developed that formulates the architecture and data flow of the accelerator, estimating and quantifying the accelerator performance and the accelerator resource utilization.
- To address the computing efficiency of DNN models in the Natural Language Processing domain, the LSTM and Transformer accelerators on the FPGA are proposed. Specifically, the proposed accelerators make the following contributions:
  - A unified computing pattern with its associated hardware optimization is proposed to enable high computing parallelism in the LSTM.
  - An algorithm-hardware co-design for Transformer is proposed to prune the algorithm and enable efficient sparse Transformer acceleration on FPGA.
  - The analytical model is developed to determine accelerator parameters and running schedule, and predict system performance for LSTM and Transformer accelerator.
- To efficiently map a multi-modal multi-task DNN model to a multi-accelerator system, a mapping framework with both computation and communication awareness aiming at

system-level formulation, modelling, and optimization is proposed. Specifically, the heterogeneous model to heterogeneous system mapping makes the following contributions:

- The multi-modal multi-task model and the multi-accelerator system are formulated to graphs to depict the model layer dependency and accelerator execution dependency.
- A low time complexity algorithm that can accurately and quickly detect the optimized mapping solution for the multi-modal multi-task model to multi-accelerator system is proposed.
- A system-level simulator for system latency and energy simulation for multi-modal multi-task DNN model to multi-accelerator is proposed which takes customized DNN model and multi-accelerator system as input.

1.3 Dissertation Organization

This dissertation proposal is organized as follows:

Chapter 2 introduces the background of the multi-modal multi-task DNN model, CNN, LSTM, Transformer, and multi-accelerator system.

Chapter 3 proposes the FPGA accelerator design for Convolution. The corresponding architecture design, workload partition, and performance model are explored.

Chapter 4 proposes the FPGA accelerator design for LSTM and Transformer. The co-design and optimization of the model and accelerator architecture are explored.

Chapter 5 proposes the mapping algorithm of multi-modal multi-task to the multi-accelerator system. An associated system-level simulator is built for system performance formulation, simulation, and optimization.

Chapter 6 summarizes this dissertation.

2.0 Background

In this chapter, the heterogeneous model (multi-modal multi-task) with its associated DNN models in Computer Vision and Natural Language Processing domain will be presented. The heterogeneous system (multi-accelerator) will also be explored in this chapter.

2.1 Heterogeneous DNN Model

Multi-modal multi-task DNN enlarges ML the model size and model complexity by jointly running several tasks simultaneously. The overview of MMMT model is shown in Fig. 1. Different modalities such as images, videos, speech, text, etc., are processed by their corresponding modality nets first. The modality nets function as the feature extraction module in the MMMT. Next, the extracted features from different modalities can be selectively fused and then being processed by the task nets. The task nets are also task-specific while each net is generally independent of each other in the model type. In both modality net and task net, the inner model data sharing (inter-block connection) are the common strategy to enhance the MMMT accuracy, reliability, and robustness. [25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36]. However, such architecture leads to more complex DNN models with more data movement as different types of DNNs are adopted and mutual dependency is applied between them.

Figure 2 shows VlocNet [35], a real-life MMMT model for semantic visual localization. In order to conduct the vision, pose and odometry inference, images from different time steps and angles are processed in a multi-task manner. In this model, three CNN backbones are placed which consist of Residual layer, Convolution layers, De-convolution layers, warping layers, and Fully-connection layers. The inner model dependencies are seen in each pair of the Residual blocks (Res). Though three backbones are in similar architecture, their layer size and model depth is customized according to their task.

Figure 3 [31] shows a real-life MMMT model to co-process image and radar signal in an autonomous driving, the image signal is processed by a VGG16 [37] like backbone, in which,

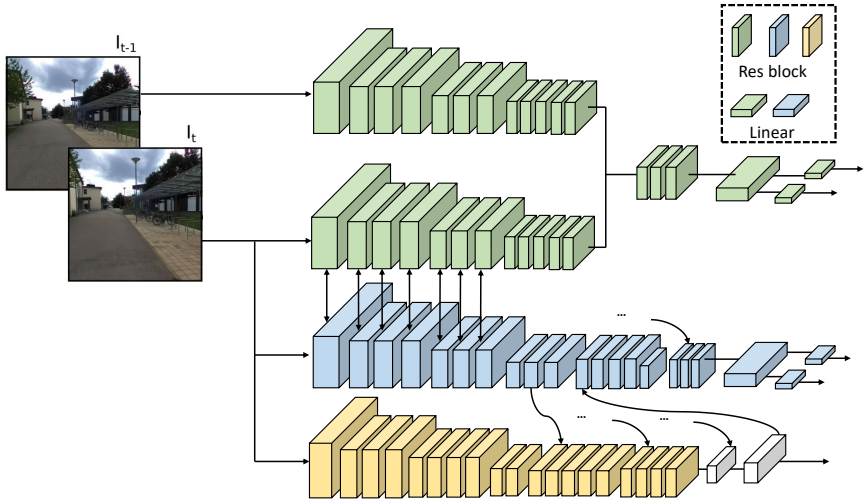


Figure 2: VLocNet++:semantic visual localization.

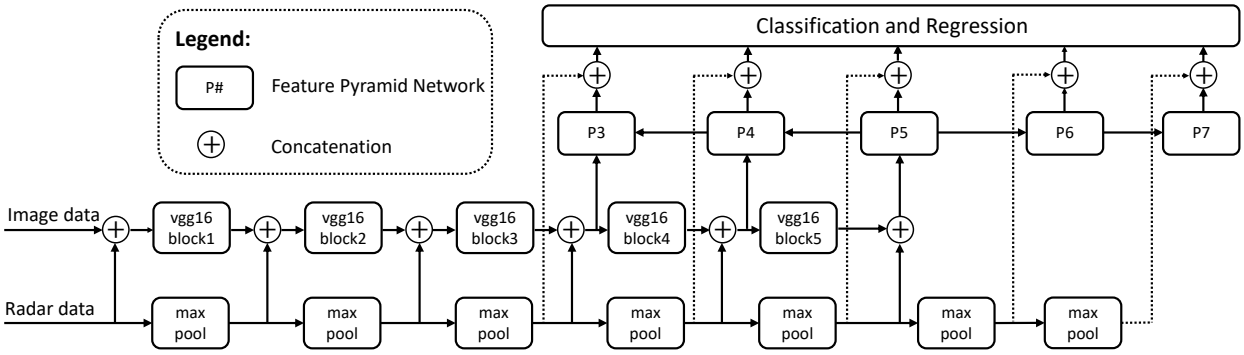


Figure 3: MMMT model for Camera-Radar data analysis.

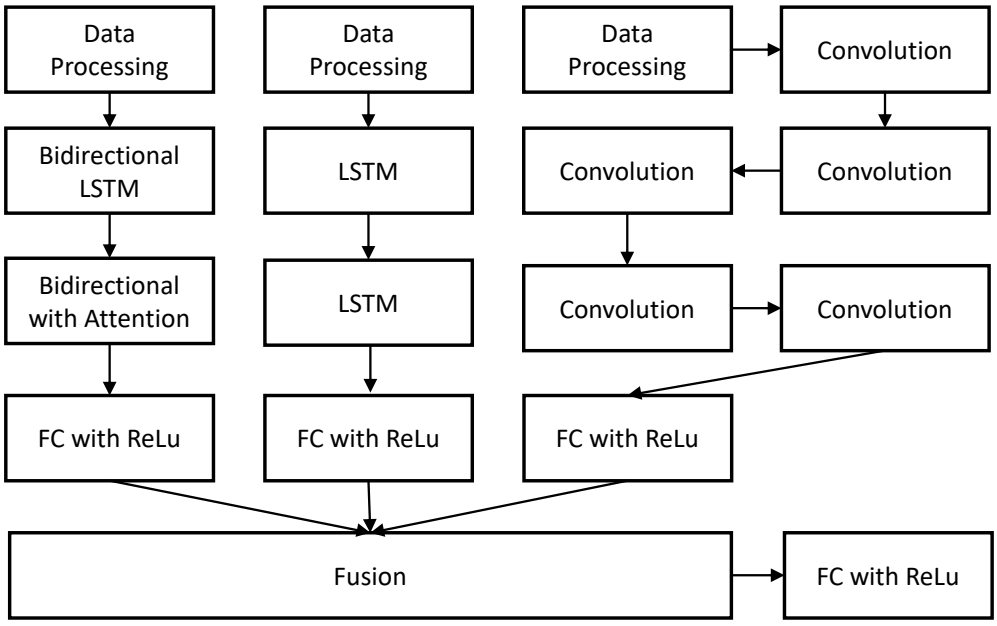


Figure 4: MMMT model for Emotion Recognition.

the VGG16 blocks are Convolution based modules. The radar signals are processed by a series of MaxPool layers. The extracted radar signals are concatenated to the intermediate data of image processing at the early stage. When going deeper, the image features and radar features are jointly processed by Feature Pyramid Network (FPN) [38]. In FPN, Convolution layers are adopted to down-sample the features. The final actions are determined by classification and linear regression steps which are realized by FC layers and SoftMax modules.

Figure 4 [36] shows an MMMT model for emotion recognition, in which, Convolution layers and LSTM cells are adopted to analyze the feature of speech, text, and motions. The Convolution layers in this MMMT model are specified in kernel size and the number of filters. The data fusion happens at the end of each sub-model. Similar to other MMMT models, different types of DNN layers are adopted in this model to process different modalities but the data sharing happens at the end of the model.

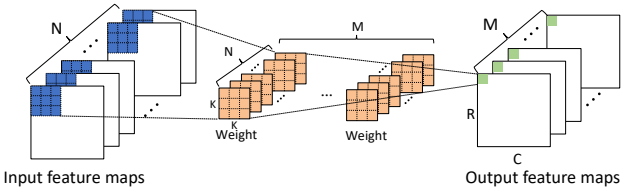
As can be seen in the real-life MMMT models, the multi-modal multi-task leads to diverse DNN layers. The diversity of the DNN layer rise challenges in computing efficiency since different DNN layers are unique in their computation and memory patterns. The inter-block

connection in the MMMT leads to complex layer dependency, which may lead to frequent data movement and accelerator execution dependency due to the mutual dependency between the layers in the heterogeneous model.

2.2 Convolution in Computer Vision Domain

CNNs are widely used as the model backbone in processing Computer Vision based tasks. Popular CNNs such as AlexNet [39], VGG16 [37], ResNet [40], YoloNet [41], etc., are the record breakers in the past years of the vision field. Among all the CNNs, they share the same core function: Convolution. A convolution layer in CNN is shown in Figure 5, which consists three parts: input feature maps (IFM), weights (WEI), and output feature maps (OFM). The output feature maps are the filtered results of input feature maps. The Convolution layer uses weights to filter the input feature maps, in which, one weight filter can extract the features across all the input feature map channels and generate one output feature map. Figure 5a shows a CNN layer with N IFM channels, M weight filters, and M OFM channels. To generate one pixel in an OFM channel, each input feature map is convolved by a shifting window in size of K $\times$ K and the convolved results from all IFM channels are accumulated, in which, the sliding window size is consistent with weight filter size K [14, 42, 43]. The depth of each weight filter is the same as the number of IFM channels N. By processing the input feature maps with M weight filters, the output feature maps with M channels are generated. The pseudo-code of a convolution layer is summarized in Fig. 5b. In general, a CNN network is composed of Convolution and other auxiliary down-sampling layers such as Pooling layers. The output feature maps of the last CNN layer are usually low-dimensional vectors that contain the extracted features. Such feature maps will be further fed into FC layers to infer the relevancy between source data and a target which is shown in Figure 6 . The architecture of FC is shown in Figure 6a, where a vector is processed by a sequence of the feed-forward neuron. In an FC layer, elements of the input vector are directed to neurons in the FC layer respectively. An edge in the FC architecture denotes a weight parameter. For a neuron, its output can be summarized as $y_1 = f(w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4)$ [44, 45]. Therefore,

the computations in an FC layer can be summarized as matrix multiplication which is shown in Figure 6b. In practice, the input to an FC layer are vectors, and the number of neurons is reflected by the number of weight columns. Convolution and FC are usually adopted as the backbone of DNN models when dealing with different CV tasks while the aforementioned auxiliary modules are embedded in the backbone for better model performance.

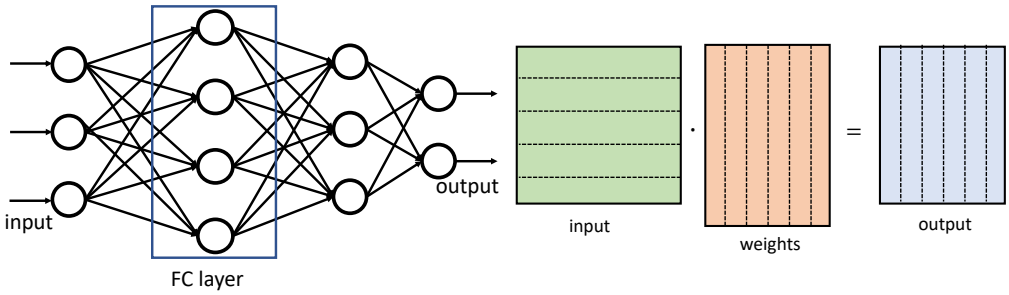


(a) CNN feature maps and weights

```
For ( row=0; row<R; row++ ) {
  for ( col=0; col<C; col++ ) {
    for ( to=0; to<M; to++ ) {
      for ( ti=0; ti<N; ti++ ) {
        for ( i=0; i<K; i++ ) {
          for ( j=0; j<K; j++ ) {
            Output_fm[to][row][col] +=
                weights[to][ti][i][j]*
                input_fm[ti][S*row+i][S*col+j]
} } } } } }
```

(b) CNN nested loops

Figure 5: A convolution layer and its computing Pseudo code



(a) FC neurons.

(b) FC Computations.

Figure 6: Fully connected layer breakdown.

2.3 LSTM and Transformer in Natural Language Processing Domain

Long Short-term Memory (LSTM) [46] and Transformer [47] are the most popular DNN models in Natural Language Processing domain due to their advantage in contextual data

analysis. LSTM outperforms other DNN models due to its unique gate architecture for data filtering and memorizing. Transformer outperforms other DNN models due to its multi-head self-attention for contexts correlation. In this section, the two models will be introduced.

2.3.1 Long Short-term Memory

Long Short-term Memory (LSTM) network is a branch of the Recurrent Neural Network (RNN) which utilizes matrix-multiplication as the internal operations. LSTMs are promising models in analyzing the data's context. The cell shown in Figure 7 is the smallest unit in the LSTM network where its input is x_t and output is h_t. LSTM cells can also be stacked to build a deeper LSTM network if needed where the upper cell takes the lower cell's output as input and the top cell's output is the network output. The cells in LSTM networks are identical in operations and each cell consists of input gate i_t, forget gate f_t, and output gate o_t, corresponding to input data remember, input data forget, and input data to current inference. LSTM takes a sequence of data $X = (x_1, x_2, ..., x_t)$ as input and processes the element x_t (a vector) in order, generating output $Y = [h_1, h_2, ..., h_t]$. During the inference of x_t, LSTM also takes x_{t-1}'s inference output h_{t-1} and cell state c_{t-1} as the auxiliary inference input which is represented by the two backward connections in Figure 7. In general, x_t is an embedded vector $[x_1, x_2, ..., x_D]$ in length D and i_t, f_t, o_t, $\widetilde{c}_t$, c_t, h_t results are vectors in length H likewise, where H is also the number of hidden states in LSTM. When extracting features in different gates, the matrix-multiplication based operations are adopted.

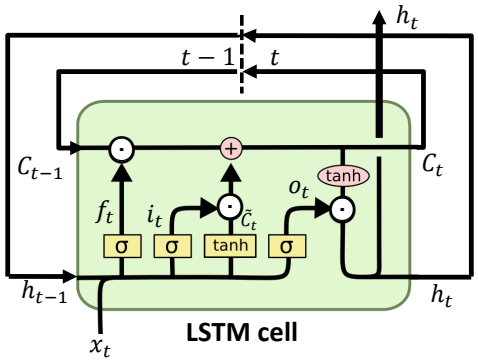


Figure 7: LSTM architecture.

$$
\begin{aligned}
i_t &= Sig(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \\
f_t &= Sig(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \\
o_t &= Sig(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \\
\hat{c}_t &= tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \hat{c}_t \\
h_t &= o_t \odot tanh(c_t)
\end{aligned}
\tag{2.1}
$$

The input x_t is processed by gates i_t, f_t, and o_t first, which is shown in Equation 2.1. The three gate-process have same patterns of computation which are matrix-vector multiplications ($W_x x_t$, $W_h h_{t-1}$) and element-wise vector addition, where $W_x x_t$ is of size $[H \times D] * [D \times 1]$, $W_h h_{t-1}$ is of size $[H \times H] * [H \times 1]$, and vector addition is of size $[H \times 1] + [H \times 1]$. The similar computation is also in $\widetilde{c}_t$, which is the prefix gate to refresh cell state c_t. During the four gates computing, activations such as *Sigmoid* and *tanh* are applied to the gate intermediate results. Then, the cell state c_t and inference output h_t are computed by element-wise vector multiplication and addition of size $[H \times 1] \odot [H \times 1]$ and $[H \times 1] + [H \times 1]$. Herein, the LSTM cell's main computing patterns are four-fold: $[H \times D] * [D \times 1]$ by MAC of size D; $[H \times H] * [H \times 1]$ by MAC of size H; $[H \times 1] + [H \times 1]$ by element-wise addition; $[H \times 1] \odot [H \times 1]$ by element-wise multiplication. The different computing patterns make it the obstacle when deploying LSTM network on accelerator as it prevents the accelerator system parallelism.

2.3.2 Transformer

The core of the Transformer is self-attention, which is also called scaled dot-product. The scaled dot-product is abstracted in Equation 2.2 [47], in which, an input $X^{in} \in R^{N \times d_{model}}$ is mapped to an output $O^{attn} \in R^{N \times d_k}$ via Query (Q), Key (K), and Value (V). The Q, K, and V are intermediate results that are acquired by multiplying X^{in} with corresponding weights Q^w, K^w, and V^w in the same size ($w \in R^{d_{model} \times d_k}$). Therefore, the computations can be roughly divided into two parts: the Q, K, and V mapping and scaled dot-product as shown in Figure 8. In a self-attention, N and d_{model} are determined by input dataset and d_k

is one of the model hyper-parameters that vary in different models. The X^{in} multiplies with corresponding weights to acquire Q, K, and V. Then, the intermediate result Q multiplies with the transpose of K (K^T). After division and softmax, the result QK multiplies with V to get O^{attn}. The overview of multi-head self-attention operations is shown in Figure 9. The multiple parallel instances of self-attention form multi-head self-attention. The individual head output O^{attn} are concatenated and mapped to multi-head output O^{head} via weight O^w. The main computations and the data flow of the multi-head self-attention mechanism is shown in the Figure 9 (b).

$$O^{attn} = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.2}$$

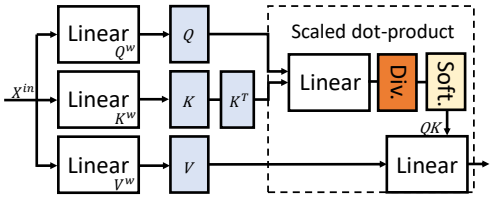


Figure 8: Computations in self-attention.

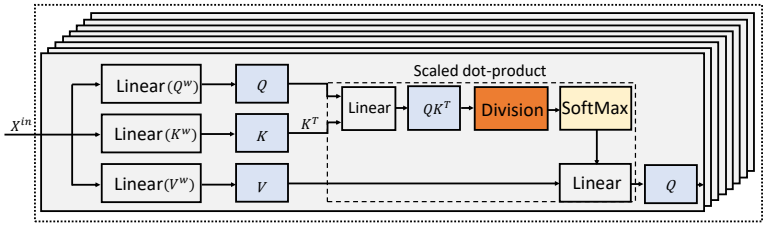


Figure 9: Multi-head self-attention.

The self-attention is further assembled as an encoder and decoder, forming the main layers of a Transformer as shown in Figure 10. In an encoder, addition_&_normalization and feed-forward network (FFN) are also placed. The FFN consists of two stacked linear modules (the same as FC). Compared to encoder in model architecture, decoder has an extra masked-attention module to build the data dependency in the output sequence. In [47], the number of heads (N_{head}) is 8, the number of the encoders (N_{enc}) is 6, and the number of the decoders (N_{dec}) is 6. The encoders are sequentially connected and decoders are connected likewise. The last encoder feeds the intermediate data into the decoders. The

input data $X = (x_1, x_2, ..., x_N)$ is embedded to X^{in} and processed by encoders first. The intermediate representation $Z^{in} \in R^{N \times d_{model}}$ generated by the last encoder is further fed into N_{dec} decoders.

As a result, the weights in a Transformer can be summarized in six types: $Q^w, K^w, V^w \in R^{d_{model} \times d_k}$, $O^w \in R^{d_{model} \times d_{model}}$, $FFN1^w \in R^{d_{model} \times 4d_{model}}$, and $FFN2^w \in R^{4d_{model} \times d_{model}}$. With the multi-head self-attention, encoders, and decoders, the number of different shapes of weights can be up to hundreds and the total memory footprint size for the Transformer is 176 MB [47]. The large size of memory requirement and various sizes of the weights make it challenging when deploying Transformer on edge devices as the memory size and computing resource are usually limited on such platforms.

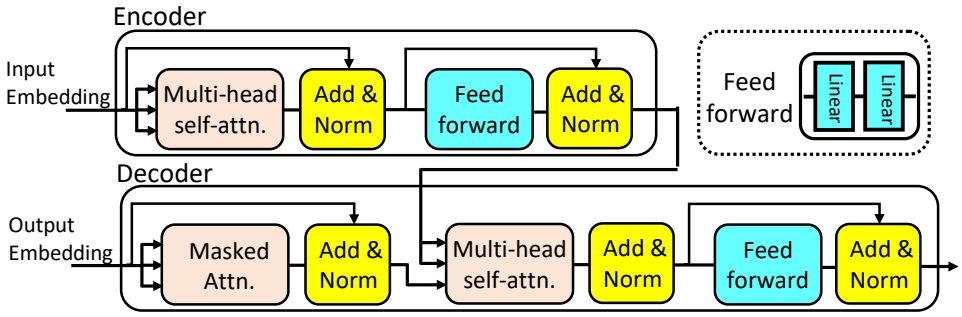


Figure 10: Transformer encoder and decoder.

2.4 Heterogeneous System

The multi-modal multi-task DNN models rise challenges in both computation and communication during the processing time. Heterogeneous computing is a promising solution to address the computing efficiency in the heterogeneous model. A DNN layer can be processed by its preferred accelerator to improve the overall system efficiency.

On the SoC level, Xilinx Versal [48], Nvidia Xavier [49], and Tesla FSD [50] integrated different accelerators on a single chip. Figure 11 shows the two state-of-the-art architectures: Tesla FSD [50] and Xilinx Versal [48]. Tesla FSD adopts a heterogeneous accelerator design which includes GPU, Neural Processing Units (NPU), and CPU cores. Xilinx Versal also adopts heterogeneous accelerator design which includes AI engines and Configurable Logic

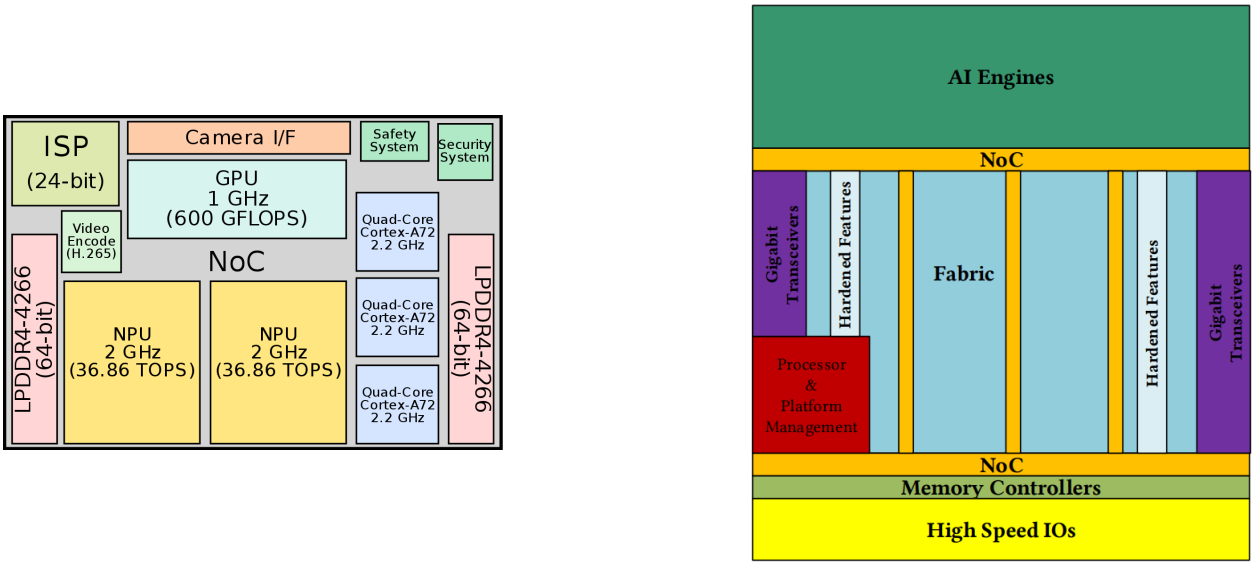
Programmable (FPGA) Accelerator. In both designs, the accelerators are linked on the chip in a network-on-chip (NoC) fashion. On a heterogeneous multi-accelerator platform, computing tasks can be assigned to different accelerators according to the task type, task complexity, and workload. Such architecture largely increases the computing efficiency when dealing with unbalanced workloads.

On the system-level, Microsoft's Brainwave [5] integrated both CPU and FPGA in the cloud and AWS [6] integrated multi-FPGA in their cloud. [7] proposed two-FPGA PL directed connection to achieve near double throughput performance. [8] proposed a workload sharing strategy between GPUs and FPGAs. [9] proposed a multi-FPGA framework in accelerating a CNN. [10] proposed a GPU-FPGA framework for the DNN training and inference process. [11] proposed a CPU, GPU, and FPGA cooperative framework for better computing performance. While most explorations adopted PCIE as the connection between accelerators, Microsoft's Brainwave [5] adopts network connection among the accelerator to achieve maximum system flexibility. The topology of FPGA based cloud system in [5] is visualized in Figure 12. Each FPGA can be customized to a specific computing kernel with dedicated data flow which achieves maximum computing efficiency for the targeting workload. By assigning the tasks from the host to its preferred FPGA accelerators, the whole system can get maximum computing efficiency. In this dissertation, we adopt a network connection for the heterogeneous accelerator which is similar to [5] to ensure the diversity and flexibility of the multi-accelerator system.

FPGAs are re-configurable devices that outperform GPUs and CPUs in energy efficiency and ASICs in flexibility. Due to its nature of highly configurable and energy-efficiency architecture, FPGAs become the promising platform for accelerator prototyping to keep pace with the rapid evolution of ML algorithms. With the emerging High-level Synthesis (HLS) tool, the FPGA design time is greatly shortened from days of RTL design to hours of high-level language programming such as C++ and OpenCL [51, 52]. Such intrinsic features make FPGAs the promising platforms to keep pace with the rapid development of the machine learning market. However, as the hardware accelerator, FPGAs also face challenges when processing DNN models. The architecture of modern FPGAs limits its on-chip memory (L1 in Figure 12) size which is usually dozens of Megabits (Mb) which the DNN models usually

requires hundreds of MegaBytes (MB) or even Giga Bytes (GB) [37, 47, 39, 40, 53, 41]. This leads to the requirement of careful design that utilizes both FPGA on-chip resources and its associated off-chip DRAM. Therefore, the computing pattern and the data flow of the FPGA accelerator is usually specialized for a pair of FPGA and its targeting DNN [12, 14, 15, 16, 17, 54, 55, 56, 57, 58, 59]. In this dissertation, we adopt FPGAs as the accelerator prototyping platforms to explore the suitable architecture for different DNN computations.

While the heterogeneous multi-accelerator system can greatly increase the computing efficiency. The transmission between the accelerators can easily become the system bottleneck as the accelerator to its local DRAM is usually much faster than the inter-accelerator bandwidth. For example, the FPGA U280 to its local DRAM bandwidth has reached 460 GB/s [60] while its out of the board bandwidth via PCIE can only achieve 15.8 GB/s; the GPU A100 to its local DRAM bandwidth has reached 2039 GB/s while its out of the board bandwidth 'NVLink' achieves 600 GB/s and PCIE achieves 64 GB/s [61]. A significant gap lies in between the accelerator to its local DRAM bandwidth and out of the board bandwidth. While the inner-block connections in the multi-modal multi-task DNN are widely applied, the cross-accelerator communication can not be avoided. Therefore, a mapping of multi-modal multi-task DNN to multi-accelerator with the consideration of the trade-off between computing and communication is vitally needed.



(a) Tesla FSD chip

(b) Xilinx ACAP chip

Figure 11: The state-of-the-art multi-accelerator chip

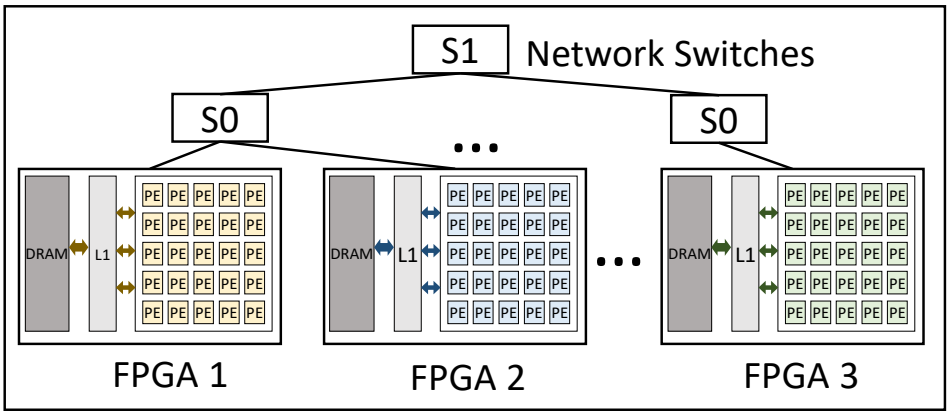


Figure 12: Multi-FPGA system.

3.0 Accelerator Optimization for Convolution Neural Networks (CNN)

This chapter presents an accelerator design that addresses the computing efficiency of Convolution on FPGA [62]. It is organized as follows. First, the background of this project is introduced, and then the motivation is presented. Next, the details of the proposed techniques are presented including the accelerator architecture design and performance modeling. Finally, the experimental results are presented.

3.1 Background

CNNs are primarily used in vision-related tasks to conduct feature extraction. A CNN usually consists of sequentially connected Convolution layers, Pooling layers, ReLu, and Normalization to down-sample the image data. The last Convolution layer is usually followed by FC layers to further reduce the model output size. Popular CNNs such as AlexNet [39], VGG16 [37], ResNet [40], YoloNet [41], etc., adopt the similar repeating manner of 'Convolution-Pooling-ReLu-Normalization'. However, each model greatly differs from others in the hype-parameters, for example, the Convolution kernel size can be 3, 5, 7, 11, or even larger; the Convolution channel size can be 16, 32, 64, 128 or even larger; the Pooling layer can adopt maximum-pooling or average-pooling with different pooling window size. Among these layers, Convolution layers are the most computing and communication intensive layers which can easily exceed FPGA's on-chip capacity. This makes Convolution the main difficulty when accelerating CNN on an FPGA accelerator.

The AlexNet [39] for image classification is visualized in Fig. 13. In AlexNet, there are five-stage 'Convolution' and three FC layers. The five stages are similar in module connections and the first stage is depicted in this figure. In one stage, the input features maps are processed by Convolution, ReLu, MaxPool, and Normalization. Among such layers, the Convolution down-samples the feature map size and enlarges the activation channel size. The Convolution operation with the kernel size K*K is visualized in Figure 14. In stage 1

Convolution, with the Convolution window of 11*11, the feature map size is down-sampled from 224*224 to 55*55 while the channel number is enlarged from 3 (RGB image) to 96. Therefore, in this Convolution layer, the IFM N is 3 and the OFM M is 96, weight kernel size K is 11, resulting in a weight size of [96][3][11][11] ([M][N][K][K]). According to the Convolution computing pattern shown in Figures 5 and 14, the number of MAC operations in this Convolution layer is R*C*K*K*N*M*2, which results in 2.1E+08 number of Multiply Accumulate. The followed ReLu layer element-wise activates the feature map (55*55*96) after Convolution which is shown in Figure 15. A MaxPooling further down-samples the activation size to 27*27*96 by reserving the largest value in a sliding window (3*3 window) with a stride of 2, which is visualized in Figure 16. The five stages 'Convolution' down-sample the feature map size from 224x224 to 13x13 and up-sample the feature map channels from 3 to 256. As shown in Fig. 5b and Fig. 14, the size of a Convolution layer input feature maps (IFM), output feature maps (OFM), and weights are multi-dimension. Even a single Convolution layer's memory footprint and workload can easily exceed an accelerator's capacity [13, 14]. Further more, the modern CNN based networks are usually designed with numerous of Convolution layers and dozens or even hundreds of feature maps in a layer to improve the feature extraction ability. As can be seen in the visualized AlexNet, Convolution layers are the main workload that needs to be addressed when processing CNN on FPGA, while other layers are usually fused into the Convolution accelerator.

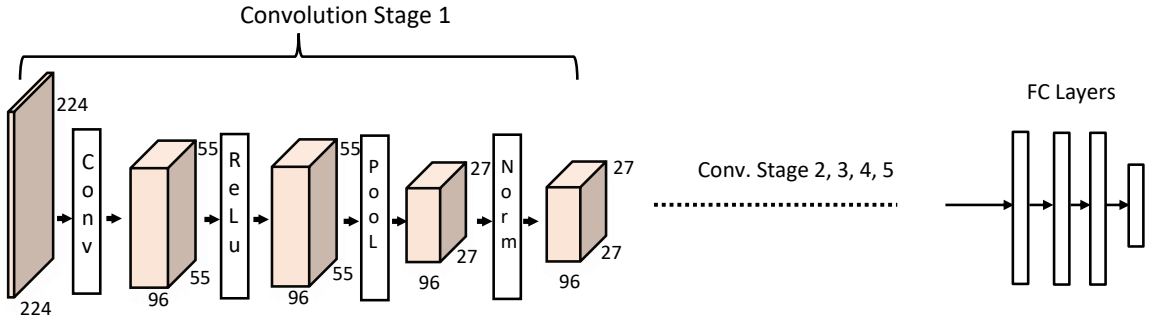


Figure 13: AlexNet architecture.

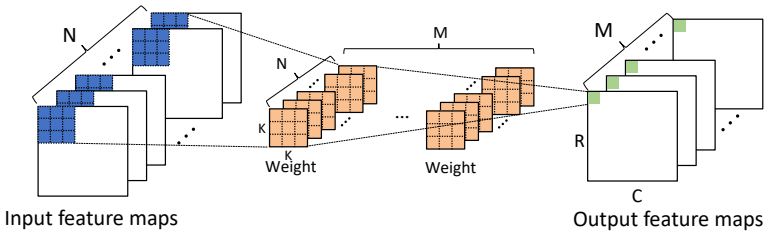


Figure 14: Convolution operation.

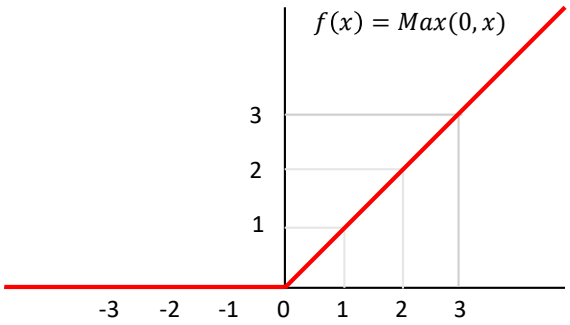


Figure 15: ReLu activation.

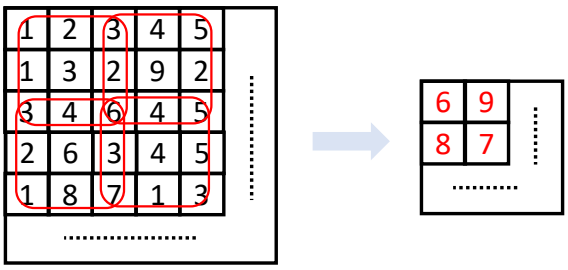


Figure 16: MaxPool.

3.2 Motivation

When accelerating the DNN algorithms, the real-time inference has rigorous requirements of guaranteed latency to ensure user experience, reliability, and even safety. The widely adopted backbone CNN is computation and transmission intensive. Its intensive computation workload and memory footprint size need to be addressed for FPGA accelerators considering the scarce FPGA on-chip resource. An efficient accelerator design and workload partition towards CNN operations is the key factor to ensure CNN processing. Figure 17 shows two popular accelerator architecture designs that is observed in the Convolution accelerators targeting a Convolution layer level processing [12, 14, 15, 16, 17, 54, 55, 56, 57, 58, 59, 63]: NVDLA [63] and Shi-diannao style [17]. The NVDLA style accelerators primarily target the computing efficiency of Multiply Accumulate (MAC) in the Convolution layer, achieving high throughput Convolution kernel. Shi-diannao style [17] accelerators primarily target on both the memory efficiency and computing efficiency. Its distributed local buffer under Processing Element (PE) can maximize the re-use of Convolution weight and its PE topology can form systolic array architecture that achieves high throughput.

Besides the accelerator architecture, the Convolution workload is usually partitioned to get processed which is shown in Fig. 18, which is loop-tiling. An OFM tile in green with size [Tm][Tr][Tc] is shown in this figure. The associated Convolution weights (orange) can be determined as [Tm][Tn][K][K] and the IFM tile (blue) can be determined as [Tn][Tr'][Tc']. By carefully determining the tiling parameters Tm, Tn, Tr, and Tc, the FPGA on-chip buffer and computing kernel can accommodate the workload of one tile. By recursively calling the accelerator to process the tiles, the workload of a Convolution layer can be processed.

However, an efficient Convolution kernel design with its performance model to determine the computing and communication efficiency is still missing. In this work, firstly, a Convolution FPGA accelerator with its accurate analytical model to quantify the performance-resource trade-off in terms of computation and communication patterns is proposed. Second, based on the FPGA accelerator and its performance model, an exploration of increasing the FPGA throughput by multi-FPGA is conducted.

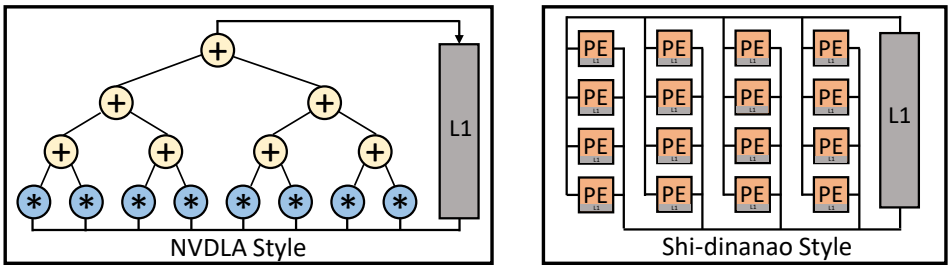


Figure 17: NVDLA and Shi-Dinanao Style Accelerator.

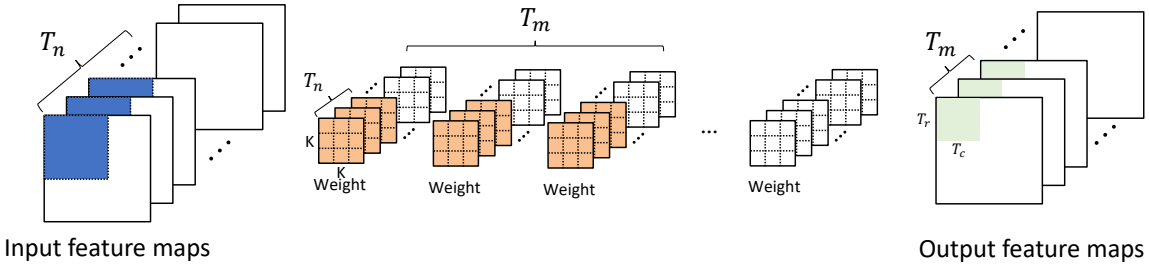


Figure 18: Convolution layer tiling.

3.3 Convolution Accelerator and Analytical Model

Accelerator overview. The overview of the proposed Convolution accelerator is shown in Figure 19. In this figure, the off-chip DRAM and the on-chip computing kernel with associated buffers are shown. The computing kernel consists of T_m Processing Elements (PE), in which, a PE performs T_n MAC via DSPs simultaneously. As a result, a total of T_m*T_n MACs are performed at the same time in the accelerator. The related PE array is visualized in Figure 20. Each PE consists of multipliers and adders, in which, the adder tree accumulates the multiplication results. Such architecture maximizes the frequency and pipeline performance of each PE and the PE array. According to the computing pattern of the PE array, the order of the Convolution loop is optimized as shown in Figure 21. Compared with the conventional Convolution loop order which is shown in Figure 5b, the loops of the Convolution sliding window are pulled to the outer loop. The OFM tiling is performed at $loop_R$ (T_r), $loop_C$ (T_c), $loop_M$ (T_m), and $loop_N$ (T_n), in which, the tiling of $loop_M$ (T_m), and $loop_N$ (T_n) can be processed via the PE array in parallel.

Besides the PE array, three buffers are allocated on the FPGA, IFM ($[T_n][T_{r'}][T_{c'}]$), WEI ($[T_m][T_n][T_r][T_c]$), and OFM ($[T_m][T_r][T_c]$). The FPGA bloc-RAMs (BRAM) are utilized to build the three buffers. In the proposed accelerator computing pattern, the IFM tiles and WEI tiles are streamed from the off-chip DRAM into the FPGA until one OFM tile is fully computed. To hide the transmission overhead between FPGA and the off-chip memory, double buffer is placed at the interface of buffer IFM, WEI, and OFM.

Convolution layer tiling summary. The volume of the data that can be accommodated by the allocated buffers and the PE array should be carefully determined, which corresponds to loop tiling and loop ordering for the original Convolution nested loop shown in Figure 5b. A tile of each input feature map, weight, and output feature map is the basic unit to be moved between off-chip and on-chip memory. The tiling of a Convolution layer can be described as $< T_m, T_n, T_r, T_c >$ corresponding to tiling parameters on OFM channel, IFM channel, OFM row, OFM column. Then, we can get the size of IFM tile to be $< T_n, T_{r'}, T_{c'} >$ (IFM $IFM[N][R'][C']$) according to the window and stride size, and weight tile to be $< T_m, T_n, K, K >$ (weight $WEI[M][N][K][K]$), where the K is the sliding window size. To accommodate the tiled Convolution layer on the proposed accelerator, the loops of the Convolution is reordered as it is shown in Figure 21. Such loop re-ordering benefits the proposed accelerator in processing Convolution layers with different Convolution window size. This is because the PE array is designed according to the parallelism of channel direction (T_m and T_n), resulting a 1*1 'Convolution' per execution. The main benefit of such design is that the Convolution window size of a layer does not affect the efficiency of the proposed accelerator.

Run-time schedule. The run-time schedule of the Convolution accelerator is shown in Figure 22, where I, W, PE, O represents the IFM tile transfer time, weight tile transfer time, PE array execution time, and OFM tile transfer time, respectively. As one OFM tile needs the multiplication and accumulation of multiple IFM and weight tiles, the OFM buffer will only off-load once while the IFM buffer and WEI buffer needs to load multiple times. With the tiling parameters $< T_m, T_n, T_r, T_c >$, the IFM and weight load time and OFM load time can be drawn. As shown in Figure 5, Figure 18, and Figure 22, a Convolution layer tiling with parameters T_m and T_n needs $\frac{N}{T_n}$ PE executions as per T_n IFM channels can only get partial

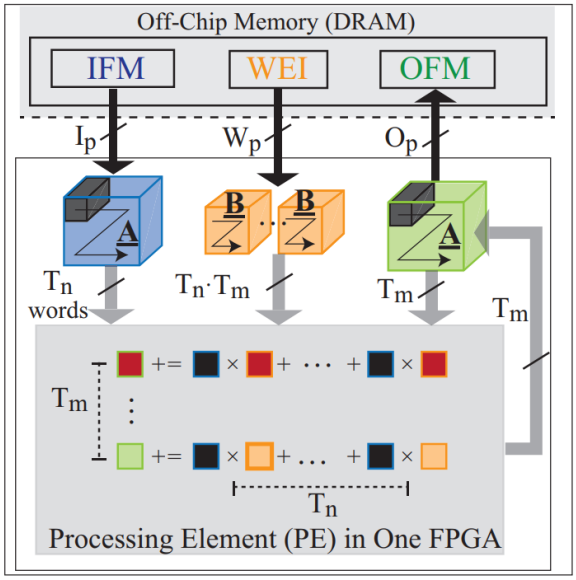


Figure 19: Convolution accelerator architecture.

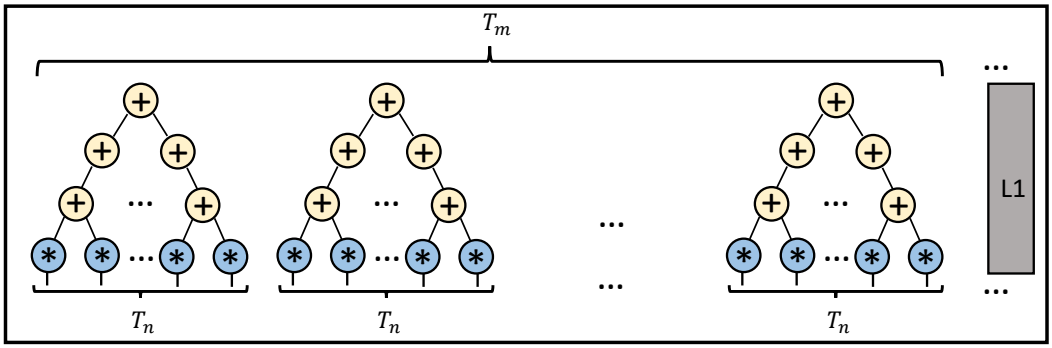


Figure 20: Convolution accelerator computing kernel.

```
for ( i=0; i<K; i++ ) {
  for ( j=0; j<K; j++ ) {
    for ( row=0; row<R; row++ ) {
      for ( col=0; col<C; col++ ) {
        for ( to=0; to<M; to++ ) {
          for ( ti=0; ti<N; ti++ ) {
            Output_fm[to][row][col] +=
                weights[to][ti][i][j]*
                input_fm[ti][S*row+i][S*col+j]
} } } } } }
```

Figure 21: Optimized Convolution nested loops.

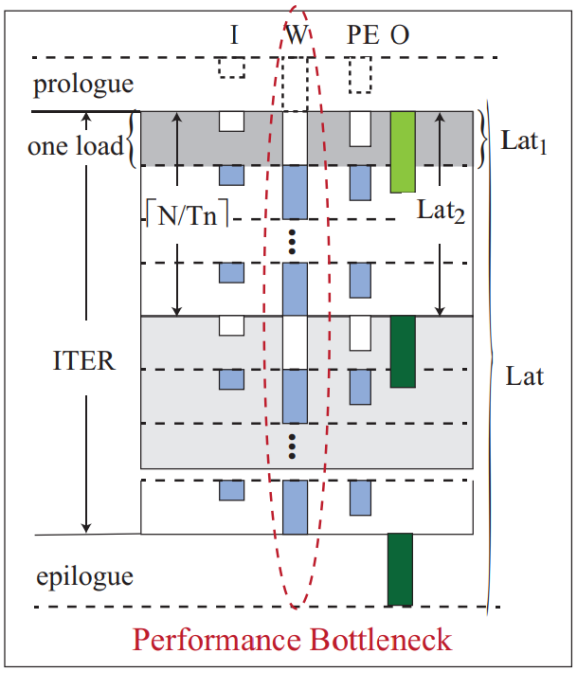


Figure 22: Convolution accelerator performance bottleneck.

results of the OFM tile. Per PE execution, a tile of IFM and weight are needed, causing IFM and WEI buffer loading. After $\frac{N}{T_n}$ PE executions, the OFM is transferred out which is shown in Figure 22. After $\frac{N}{T_n}$ executions, a OFM tile result is computed while the OFM tiling still needs to traverse along OFM channel, OFM row, and OFM column. To process the OFM tiles in channel wise, another $\frac{M}{T_m}$ PE executions are needed; to process the OFM tiles in row wise, another $\frac{R}{T_r}$ PE executions are needed; to process the OFM tiles in column wise, another $\frac{C}{T_c}$ PE executions are needed. Therefore, a total of $\frac{N}{T_n}\frac{M}{T_m}\frac{R}{T_r}\frac{C}{T_c}$ executions will be needed per Convolution layer.

Performance model. The performance model can be used to determine both of the accelerator latency and resource utilization of the design. For each PE execution, its latency is determined by the following issues: the PE number, the allocated buffer size, and the accelerator transmission performance of off-chip memory interface. The number of PEs is limited by the number of DSPs DSP_{chip} on FPGA, where the f_{dsp} is the offset of DSP utilization for different data type (e.g. $f_{dsp} = 5$ in floating point, $f_{dsp} = 1$ in fix-point). As there are T_m PE in the accelerator and each PE consists of T_n MAC operation, the PE size should fulfill the following requirement:

$$T_m * T_n * f_{dsp} < DSP_{chip} \tag{3.1}$$

The size of buffers is limited by the FPGA's on-chip buffer capacity where IFM buffer is declared as a 3-dimension array $B_{ifm}[T_n][T_r][T_c]$; OFM buffer is declared as a 3-dimension array $B_{ofm}[T_m][T_r][T_c]$; Weight buffer is declared as a 4-dimension array $B_{wei}[T_n][T_n][K][K]$. In order to support parallel data access of the PE array, these data arrays should be partitioned into different on-chip memories (i.e. BRAM) which can be accessed in parallel. As shown in Figure 19, the PE array needs the parallel access of Tn pixels in IFM, Tm pixels in OFM , and Tm*Tn weights in WEI buffers. Therefore, we completely partition IFM and OFM along their first dimension, and WEI along its first two dimensions. Then, we calculate the usage of BRAM blocks for IFM, OFM, WEI in Equation 3.2. In this buffer modeling, the f_{data} represents the bit-width of the data, $BRAM_{unit}$ represents the a unit BRAM's depth, and $BRAM$ represents the number of BRAM blocks on the chip. The allocated BRAMs for IFM, OFM, and WEI (B_{ifm}, B_{ofm}, and B_{wei}) should not exceed the FPGA's BRAM capacity. Therefore, the buffer allocation of the accelerator should follow:

$$\begin{aligned} B_{ifm} &= 2 * T_n * [T_r * T_c * f_{data}/BRAM_{unit}] \\ B_{ofm} &= 2 * T_m * [T_r * T_c * f_{data}/BRAM_{unit}] \\ B_{wei} &= 2 * T_m * T_n * [K * K * f_{data}/BRAM_{unit}] \\ B_{ifm} &+ B_{ofm} + B_{wei} < BRAM \end{aligned} \tag{3.2}$$

The accelerator transmission performance is limited by the chip physical bandwidth limit and the interface design of the three buffers. The summation of assigned bandwidth to buffer B_{ifm}, B_{ofm}, and B_{wei} should be less than the physical bandwidth limit BW which is shown in Equation 3.3. In general, the physical bandwidth is limited and supported by physical channels (e.g. 4 128Bit AXI transmission channels for ZYNQ Ultra-Scale FPGAs). As the buffer size is determined, the transmission performance for the three buffers can be modeled as in Equation 3.4.

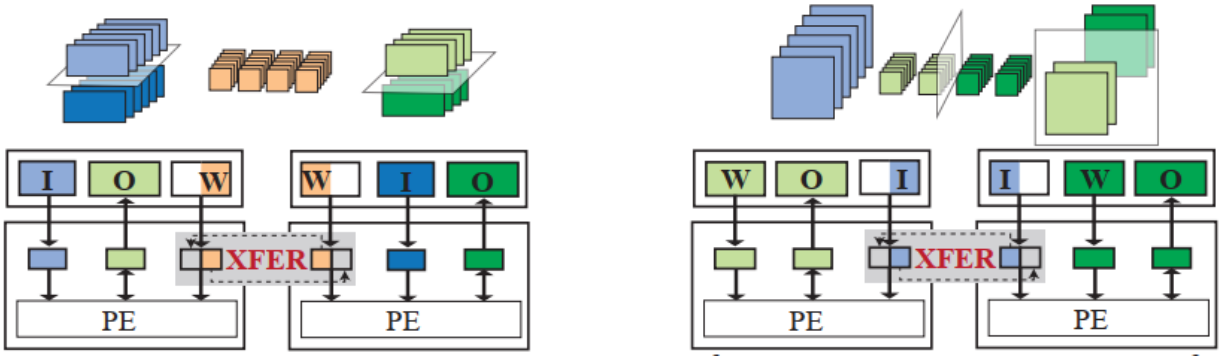
$$BW_{ifm} + BW_{ofm} + BW_{wei} < BW \tag{3.3}$$

$$
\begin{aligned}
tM_{ifm} &= T_n * T_r * T_c * f_{data}/BW_{ifm} \\
tM_{ofm} &= T_m * T_r * T_c * f_{data}/BW_{ifm} \\
tM_{wei} &= T_m * T_n * K * K * f_{data}/BW_{ifm}
\end{aligned} \tag{3.4}
$$

The latency (t_{Comp}) of one OFM tile processing is determined by the OFM tile row and column size $< T_r, T_c >$ and the weight kernel size KxK.. Therefore, according to the execution order which is illustrated in **Convolution layer tiling summary** and **run-time schedule** the latency to process one Convolution layer can be conducted as Equation 3.5.

$$
\begin{aligned}
t_{Comp} &= K * K * T_r * T_c \\
t_{PE} &= max\{t_{Comp}, tM_{ifm}, tM_{wei}\} \\
t_{Tile} &= max\{\frac{N}{Tn} * t_{PE}, tM_{ofm}\} \\
t_{Layer} &= \frac{R}{T_r} * \frac{C}{T_c} * \frac{M}{T_m} * t_{Tile}
\end{aligned} \tag{3.5}
$$

Accelerator optimization in multi-FPGA. While a layer of CNN can be partitioned to tiles to accommodate workload on a FPGA, it still exists a transmission and computing balance problem between the on-chip processing and off-chip data transmission. Due to the highly parallel design of accelerator architecture, the transmission between FPGA and the off-chip DRAM can easily cause the PE stall as shown in Figure 22, in which, weight transmission causes the accelerator stall in this example. Considering the IFM and weights sharing mechanism in Convolution, the transmission problem can be relieved by assigning workload to multiple accelerators when inter-PL (the FPGA chip is also called Programmable Logic, PL) connections with competitive bandwidth exit. The bottleneck transmission (weight in Figure 22) can be offloaded to the inter-PL connection. As shown in Figure 23, a Convolution layer can be processed by two or multiple FPGAs simultaneously. Every two FPGAs are connected by direct point-to-point communication ('XFER') such as optical fiber, in which, a FPGA fetches data from its off-chip memory and broadcast its data to other FPGAs. Meanwhile, it receives data that is broadcasted by the connected FPGAs. This benefit in less transmission time between of-chip memory to on-chip memory as long as the inter-PL



(a) Convolution workload weight sharing (b) Convolution workload IFM sharing

Figure 23: The transmission and computation workload sharing

communication bandwidth is larger than the bandwidth assigned to the bottleneck data. Taking Figure 23a as the example, a CNN layer workload is evenly split to two FPGAs according to OFM rows. Therefore, when processing the upper half and lower half OFM workload on two FPGAs, the required weights are identical for the two FPGAs. Therefore, if the weight transmision causes the accelerator stall, the weights can be split to two different parts and shared among the point-to-point communication. In such a way, the number of PEs is doubled and the transmission bottleneck is relieved, which will bring more than 2x speed as it largely reduces the computing stall.

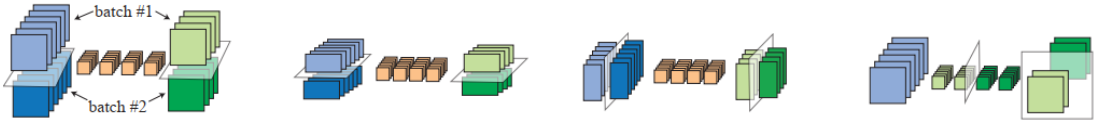


Figure 24: Two-FPGA data sharing examples.

The most common workload partition is the batch partition, where the IFM and OFM are divided along batching direction as shown in Figure 24a. The computation of a batch of OFM only relies on the corresponding batch of IFM and the whole weights. In consequence these batches can be computed in parallel in multiple FPGAs and the weights are also split to multiple parts and shared by FPGAs. Partitioning Convolution OFM along rows (R), columns (C), and channels are shown in Figure 24 b-d. The workload splitting take advantage of data sharing mechanism in CNN layer and can achieve super-linear system performance.

3.4 Experiment

3.4.1 Experiment Setup

The accelerator on FPGA is implemented with Vivado HLS, which generates design's IP core from C++ language. In HLS, we apply HLS-defined pragma to implement loop optimization. Then, the obtained IP cores are connected, synthesized and implemented in Vivado (v2017.4). In Vivado, we employ Xilinx Aurora IP core to control inter-FPGA communication and add an axi-timer to capture the exact elapsed time. FPGA boards are connected through SFP+ cables, as shown in Figure 26. Finally, we employ Xilinx SDK to program MPSoC on ZCU102, which controls the start-up of the accelerator and off-chip/on-chip communication.

The Convolution accelerator and the workload assignment is validated on a two-FPGA system consisting of two Xilinx ZCU102 FPGAs. FPGAs are connected by SFP+ optical fiber using the Xilinx Aurora IP which is shown in Figure 25. In this way, data in two FPGAs can be directly moved between their on-chip buffers. The implementation of each FPGA utilizes the ZYNQ architecture, which controls the startup of Convolution accelerator, the off-chip/on-chip communications, etc. As shown in this figure, each FPGA has two clock domains: one for accelerator and the other for board-to-board communication. We employ asynchronous FIFOs to coordinate data movements in different clock domains.

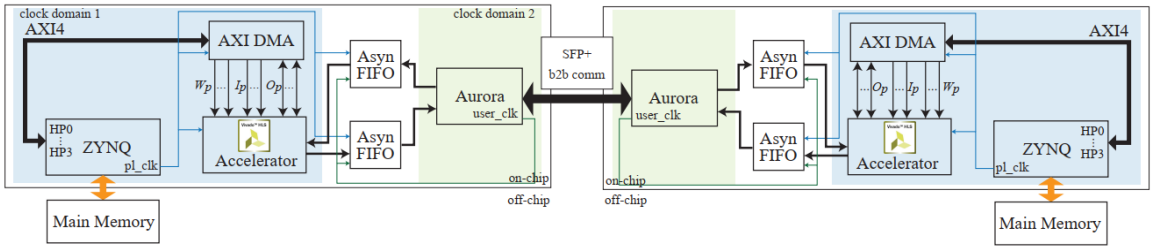


Figure 25: Two-FPGA system.

3.4.2 Performance Evaluation

Table 1 reports the comparison results in latency, throughput and energy efficiency in processing the five Convolution layers in AlexNet with a batch size of 1 on different platforms

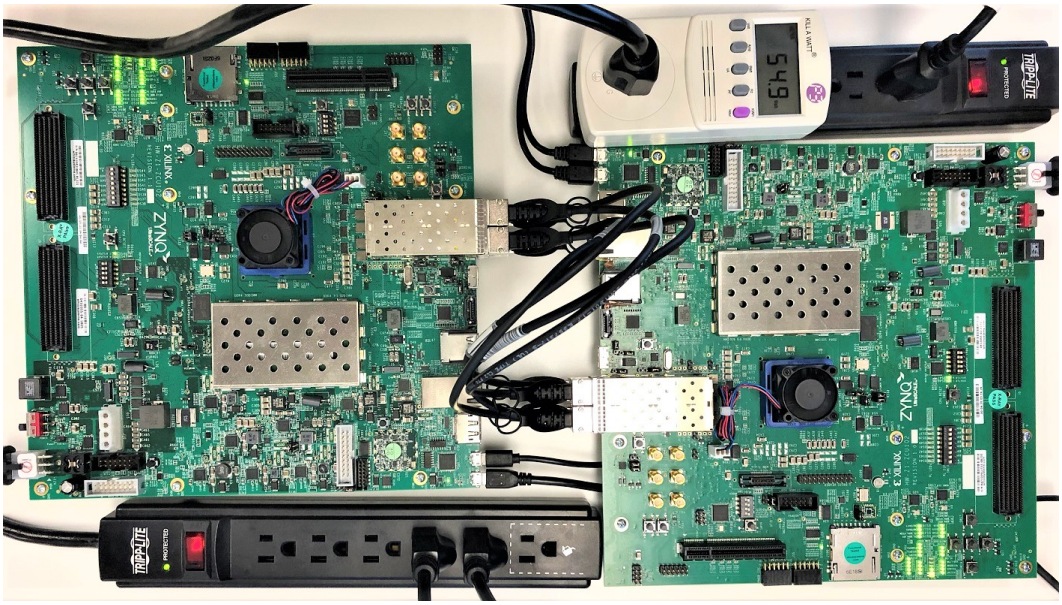


Figure 26: Power measurement of on-board executions.

and designs. The competitors include mobile GPU (Jetson TX2) and GPU (Titan X), single-FPGA design (FPGA15 [14], ISCA17 [13]), and multi-FPGA design (ISLPED16 [64]). The power consumption of our implementation is measured by a power meter as demonstrated in Figure 26. Note that notation "-" indicates that data is not reported in references or inapplicable.

Latency. Real-time DNN inference requires ultra-low latency to avoid missing deadline. For 32bits float-point, the proposed design achieves latency of 10.13ms, which is 23.26%, 2.13×, 5.94× less than that of mGPU, FPGA15 [14], and ISCA17 [13]. However, the proposed design with 32bits float-point is slower than Titan X GPU, whose latency is 6.4ms. This is because such GPU is much more powerful, with the penalty of consuming more than 3× power over the FPGA implementation in the proposed design. Benefiting from the flexibility of FPGAs to apply different data types for computation, it is possible to reduce latency by using lower-precision data type. As shown in this table, by applying 16bits fix-point, the proposed design can achieve the lowest latency among all competitors, i.e., 2.27ms.

Throughput. Compared with ISCA17 [13] with 32bits float-point, the proposed design achieves 5.94× lower latency together with 1.75× higher throughput. The improvement in throughput is less than that on latency is because ISCA17 aims to improve throughput, but

Table 1: Experimental results of 2-FPGA system with comparisons to the existing designs in accelerating AlexNet 5 Convolution layers

Design	mGPU		GPU		*FPGA15*		*ISCA17*		*ISLPED16*		**Ours**			
Precision	32bits float		32bits float		32bits float		32bits float		16bits fixed		32bits float		16bits fixed	
Device	Jetson TX2		Titan X		VX485T		VX485T		4×VX690t		2×ZCU102		2×ZCU102	
Freq (MHz)	1300MHz		1139MHz		100MHz		100MHz		150MHz		100MHz		200MHz	
Power (Watt)	16.00		162.00		18.61		-		126.00		52.40		54.40	
DSP Uti.	-		-		80%		80%		-		90.79%		55.87%	
BRAM Uti.	-		-		49.71%		43.25%		-		72.92%		92.43%	
Overall Perf.	Lat. *ms*	Thr. *GOPS*	Lat. *ms*	Thr. *GOPS*	Lat. *ms*	Thr. *GOPS*	Lat. *ms*	Thr. *GOPS*	Lat. *ms*	Thr. *GOPS*	Lat. *ms*	Thr. *GOPS*	Lat. *ms*	Thr. *GOPS*
	11.1 - 13.2	110.75	5.1 - 6.4	235.55	21.62	69.09	60.13	85.47	30.6	128.8	10.13	149.54	**2.27**	**679.04**
E.-E. (GOPS/W)	6.88		1.45		3.71		-		1.02		2.85		**12.48**	

its throughput is still less than the proposed design. Similarly, compare with ISLPED16 [64] with 16bits fix-point, the proposed design achieves 13.48× lower latency together with 5.27× higher throughput. Benefiting from the higher throughput, the proposed design achieves the highest energy efficiency than competitors.

Performance breakdown. The performance breakdown of the two-FPGA system when for the AlexNet's 5 Convolution layers is listed in Table 2. In this comparison, the performance of floating point accelerator and fix point accelerator are compared with the baseline design FPGA15 [14]. As shown in the table, for each layer of AlexNet, our design achieves over 2x speedup when compared with the baseline.

Performance model accuracy and effectiveness. The accuracy and effectiveness of the proposed system-level model is shown in Figure. We will conduct two sets of experiments: (1) we compare the proposed model with the existing one in predict latency in Figure 27; (2) we compare the proposed model with the final implementation results from Vivado in memory resource, computation resource, and on-board execution latency in Table 3.

Figure 27 reports the comparison results among different models and on-board execution latency. The x-axis and y-axis represent different designs and latency in clock cycles, respectively. In the first three designs, we employ one FPGA for implementation; while for the fourth one, we employ 2 FPGAs.

Table 2: Comparison results on 2-ZCU102

Design	32bits float				16bits fixed			
	FPGA15		**Ours**		*FPGA15*		**Ours**	
$\langle T_m, T_n \rangle$	$\langle 64, 7 \rangle$		$\langle \mathbf{64}, \mathbf{7} \rangle$		$\langle 64, 24 \rangle$		$\langle \mathbf{128}, \mathbf{10} \rangle$	
Power (W)	25.70 (1 FPGA)		**52.40 (2 FPGAs)**		26.00 (1 FPGA)		**54.40 (2 FPGAs)**	
Perf.	Lat. ms	Thr. GOPS	**Lat. ms**	**Thr. GOPS**	Lat. ms	Thr. GOPS	**Lat. ms**	**Thr. GOPS**
conv1	7.36	28.6	**3.66**	**57.6**	3.74	56.5	**0.94**	**224.5**
conv2	5.20	86.1	**2.55**	**175.5**	1.48	302.6	**0.48**	**933.1**
conv3	4.50	66.4	**1.73**	**172.7**	1.20	249.6	**0.33**	**906.2**
conv4	3.41	65.7	**1.31**	**171.0**	0.89	252.6	**0.35**	**640.8**
conv5	2.28	66.0	**0.88**	**170.9**	0.59	251.7	**0.17**	**879.5**
overall	22.75	66.6	**10.13**	**149.5**	7.90	195.1	**2.27**	**679.0**
Perf. Impr.	1.00×		**2.25×**		1.00×		**3.48×**	
E.-E. (GOPS/W)	2.59		**2.85**		7.51		**12.48**	
E.-E. Impr.	-		**9.21%**		-		**39.86%**	

Results in Figure 27 show that the latency predicted by our proposed model is always close to the on-board execution latency, where the average deviation is only 2.53%. In contrast, the existing model in [14] has larger deviations on designs of $\langle 10, 22 \rangle$ and $\langle 8, 32 \rangle$ (T_m, T_n), which are 18.49% and 45.47%.

We have another observation from Figure 27. For the design of $\langle 12, 16 \rangle$, model in [14] predicts the same latency with ours. This is because the computation latency dominates the whole system. In this case, the inaccurate estimation of communication will not affect prediction accuracy. However, when we employ more computation resource (by increasing $T_m \times T_n$), the performance bottleneck moves to communication which leads to the large latency deviations between the existing model and the on-board execution.

The above results verify the accuracy and effectiveness of the proposed system-level model in predicting system latency. With such an accurate model, it can help designers to get the accurate system performance to make better design decisions.

Table 3 reports the comparison between the proposed model and the final implementation results from Vivado in BRAMs and DSPs. It is clear that the deviations on BRAM and DSP usages are less than 7.5% and 3.9%, respectively. These deviations are mainly caused by the overhead on extra operations besides the accelerator itself, such as DSPs used for address calculation. The above results further verify the accuracy of the proposed model.

3.5 Summary

The proposed Convolution accelerator with the associated performance model is a promising accelerator that can be employed in accelerating Convolution layers on FPGA. The efficient Convolution engine design and the accuracy analytical model can be flexibly applied to a given set of Convolution layers and FPGA pair, achieving accurate and fast accelerator prototyping. The explored multi-FPGA design can achieve super-linear performance gain when there are fast connections between FPGAs.

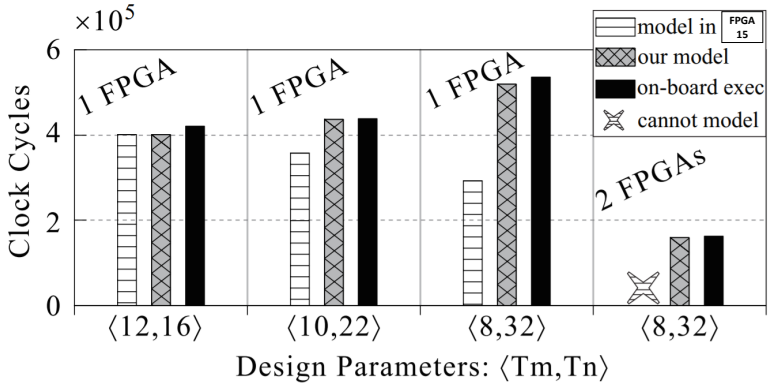


Figure 27: Comparisons of predictable models and on-board executions on latency: employing different designs on single-FPGA and two-FPGA systems.

Table 3: Convolution accelerator performance model accuracy analysis.

Design	Precision	$\langle Tm, Tn \rangle$	Partition	Our Model				On-Board			**Deviation**			Speedup
				Cycles	BRAM	DSPs	**Bound**	Cycles	BRAM	DSPs	**Cycles**	**BRAM**	**DSPs**	
A (Single)	32b float	$\langle 8, 32 \rangle$	-	519168	592	1280	**IFM**	535530	624	1326	**3.06%**	**5.13%**	**3.47%**	**baseline**
B (2-FPGA)		$\langle 8, 32 \rangle$	ofm	158880	592	1280	**Comp.**	162114	640	1331	**1.99%**	**7.50%**	**3.83%**	**3.30X**
C (Single)	16b fixed	$\langle 64, 20 \rangle$	-	115200	1448	1280	**Weight**	118688	1516	1324	**2.94%**	**4.49%**	**3.32%**	**baseline**
D (2-FPGA)		$\langle 64, 20 \rangle$	row	32760	1448	1280	**Comp.**	34622	1530	1330	**5.38%**	**5.36%**	**3.76%**	**3.43X**

4.0 Software/Hardware Co-design for LSTM and Transformer Network

This chapter presents the accelerator designs that address the computing efficiency of DNNs in the Natural Language Processing domain. It is organized as follows. First, the optimization of LSTM computation and its associated FPGA accelerator [65] is presented. Second, the Transformer compression with its associated FPGA accelerator [66] design is presented.

4.1 Accelerator Design for Long Short-term Memory (LSTM)

4.1.1 Background

As it is shown in Figure 7 and Equation 2.1, a LSTM cell includes four gates: input gate (i_t), forget gate (f_t), output gate (o_t), and cell state gate ($\hat{c}_t$). The data flow of the four gates can run in parallel and they share the same computing pattern:

① $[H \times D] * [D \times 1]$ by MAC of size D.

② $[H \times H] * [H \times 1]$ by MAC of size H.

③ $[H \times 1] + [H \times 1]$ by element-wise addition.

④ $[H \times 1] \odot [H \times 1]$ by element-wise multiplication.

In the summarized computing pattern, the H represents the LSTM model hidden size and the D represents the LSTM input vector length. The pattern ① is visualized in Fig. 28; the pattern ② is visualized in Fig. 29; the pattern ③ is visualized in Fig. 30 (a); the pattern ④ is visualized in Fig. 30 (b). As depicted in the figures, the computations in the LSTM cell are mainly matrix-multiplications, element-wise multiplication, and element-wise addition. Therefore, each of the four gates i_t. f_t, o_t, and $\widetilde{c}_t$ along with c_t and h_t contains four different computing patterns. The complex patterns can be realized by building different computing cores in the hardware. The activations are usually processed in software or approximately computed in hardware. The eight weight-matrix W_x and W_h are independent of each other

but are recurrently used at different times. The cell state c_t and output h_t are also refreshed at different times. The FPGA devices with an on-chip buffer can take advantage of such features, buffering the weights and cell state on the chip according to its buffer capacity.

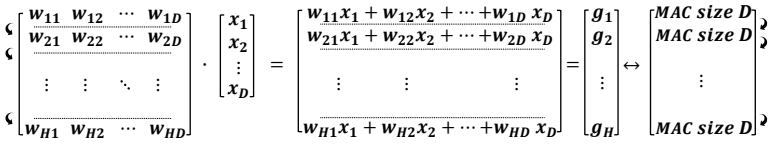


Figure 28: The matrix-vector multiplication for $W_x x_t$.

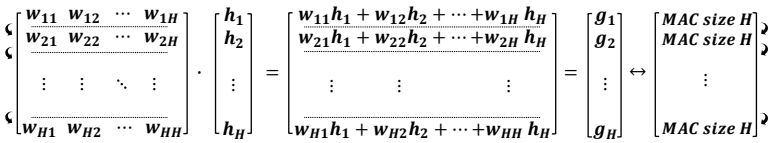


Figure 29: The matrix-vector multiplication for $W_h h_{t-1}$.

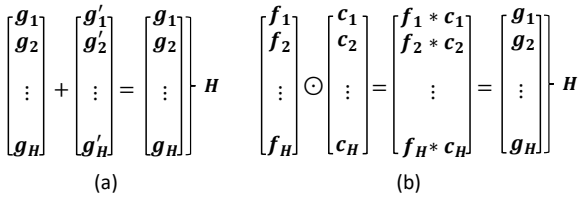


Figure 30: The element-wise addition and multiplication.

To accelerate the four gates in the LSTM, heterogeneous-kernel accelerators are proposed [19, 67]. Different MAC computing cores which consist of multipliers and an adder tree are utilized to fit pattern ① and ②. An example of the MAC computing core is shown in Fig. 31 (a). The accelerators of LSTM design kernels for gates and the rest element-wise operations [19, 67] which is shown in Fig. 31 (b). In the gate kernel, two different MAC computing cores are placed to mimic pattern ① and ②; a vector adder is placed for element-wise addition; an activation approximation is placed for *Sigmoid* or *tanh*. The adder is idle until both MAC cores finish. A dedicated kernel is designed for element-wise multiplication, addition, and activation approximation (mimic c_t and h_t). The dedicated kernel is idle until all gates computations are done. The intra-kernel and inter-kernel data dependency are shown in Fig. 32. Such architecture leads to kernel stall and low resource parallelism. Chang et al.[67] achieve 142MHz working frequency and 22.73% computing resource utilization; Guan et al.[19] achieve 150MHz working frequency and 42% computing resource utilization.

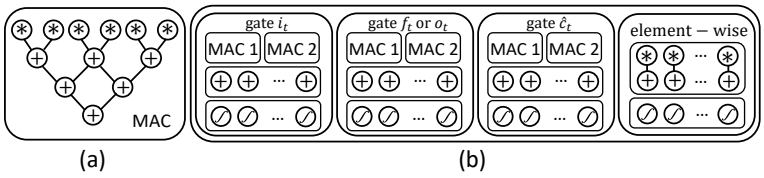


Figure 31: The computing kernels of LSTM accelerator.

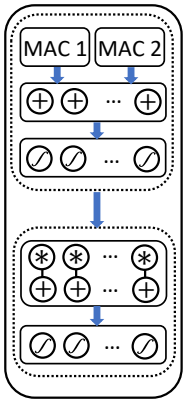


Figure 32: The data dependency of computing kernels in the LSTM accelerator.

4.1.2 Motivation and Contribution

How to fully exploit the parallelism in LSTM and maximize resource utilization is the main concern when deploying LSTM on the accelerator. Furthermore, LSTM contains heterogeneous computing patterns and data dependency among operations, which makes the problem more challenging. The naive solutions [19, 20, 67] that design the dedicated computing kernel to fit computing patterns shows unbalanced resource allocation and costly kernels stall, where device resource utilization in all the above works is less than 50% and system running stalls are observed. Without fully exploiting the parallelism in LSTM, these designs suffer large latency, which cannot satisfy the real-time requirement.

Besides computing resources, the memory in edge devices is also limited. On the other hand, LSTM has a large memory requirement for storing the model (i.e., weights). With the growing complexity of applications, the LSTM model size (e.g 3.16 Mb−173.5 Mb [68, 69, 70, 71]) is becoming bigger and may easily exceed the on-chip buffer capacity. In addition, there is a lack of efficient on-chip buffer management, leading to computing kernel idle to wait for

reading weights since the latency of costly data transfer can easily exceed the execution time [12, 72, 73, 74, 75, 76, 62, 77]. Efficient buffer management can keep weights on the chip to significantly alleviate the performance bottleneck on data transfer between off-chip and on-chip memory.

Though the data flow of the LSTM is carefully designed in the existing works, the kernel stall and low resource parallelism problems are still unsolved and leave space for further improvement. Instead of reducing the number of computations, this work reduces the number of computing patterns but keeps LSTM integrity, which will only need single-type kernels in the accelerator. With the unified kernels, the on-chip buffer management can also be more efficient.

In this chapter, a novel FPGA-based LSTM accelerator design is proposed which incorporates a unified computing kernel to simultaneously execute operations in all LSTM gates. The unified accelerator design is motivated by the observation that all operations in LSTM can be conducted by two fundamental computing patterns, the element-wise multiplication and element-wise addition. In this way, the resources in accelerators can be fully utilized, and all operations are conducted in a full parallel way. In addition to the optimization of computation, we also propose associated buffer management to maximize the device buffer utilization and in turn reduce communication latency. Finally, based on the computing engine design and buffer management, an analytical model is formulated to determine the optimal accelerator hyper-parameters, schedule accelerator operations, and predict the overall system performance.

4.1.3 LSTM Accelerator and Analytical Model

4.1.3.1 Unifying the Computing Patterns

Each LSTM gate originally requires two types of MAC and one element-wise vector addition. However, the weight matrix W_x and W_h share the same parameter H, which is the number of rows of the weight matrix as shown in Figure 28 and Figure 29. The element-wise vector addition is of size H as well. For the non-gate c_t and h_t, the computations are mere element-wise vector multiplication or addition of size H as shown in Figure30 a and b.

Taking advantage of the same number of matrix rows, the MAC operations for matrix-vector multiplications can be replaced with unified element-wise multiplication followed by addition. Instead of computing g_H element in order, all elements in $[g_1, g_2, ..., g_H]^T$ can be partially computed by multiplying a column of weight with corresponding x_t element simultaneously. The partial result (P_d) of $[g_1, g_2, ..., g_H]^T$ for $W_x x_t$ computed at column d can be denoted by:

$$P_d = [w_{1d}, w_{2d}, ..., w_{Hd}]^T \odot [x_d, x_d, ..., x_d]^T \tag{4.1}$$

As shown in Figure 33, by accumulating vector P_d from column one to column D, the full result $[g_1, g_2, ..., g_H]^T$ for $W_x x_t$ can be incrementally computed:

$$P_{x_t} = \sum_{d=1}^{D} P_d \tag{4.2}$$

Herein, as shown in Figure 34, the computation for $W_h h_{t-1}$ can also be acquired similarly:

$$P_h = [w_{1h}, w_{2h}, ..., w_{Hh}]^T \odot [h_h, h_h, ..., h_h]^T \tag{4.3}$$

$$P_{h_{t-1}} = \sum_{h=1}^{H} P_h \tag{4.4}$$

As shown in equations 4.1 to 4.4, the two sizes of MAC for $W_x x_t$ and $W_h h_{t-1}$ are replaced with element-wise multiplication and addition in the same size H. As such, the gates computation i_t, f_t, o_t, and $\tilde{c_t}$ before activation can be summarized as:

$$G = \sum_{d=1}^{D} P_d + \sum_{h=1}^{H} P_h + Bias \tag{4.5}$$

The fundamental computing patterns for a gate is further shown in Figure 35. MACs and addition are replaced with element-wise vector multiplication and addition. As the c_t and h_t are originally computed by element-wise multiplication and addition, pattern ① and

② are eliminated in the LSTM. The computing patterns in an LSTM can be summarized as an element-wise vector multiplication of size H which is shown in equation 4.6; element-wise vector addition of size H which is shown in equation 4.7. Via the two fundamental patterns 4.6 and 4.7, the gate operation (Equation 4.5) can be fully processed.

$$[v_{11}, v_{12}, ..., v_{1H}]^T \odot [v_{21}, v_{22}, ..., v_{2H}]^T \tag{4.6}$$

$$[v_{11}, v_{12}, ..., v_{1H}]^T + [v_{21}, v_{22}, ..., v_{2H}]^T \tag{4.7}$$

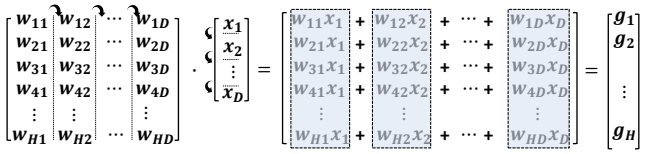


Figure 33: Applying fundamental patterns to $w_x x_t$.

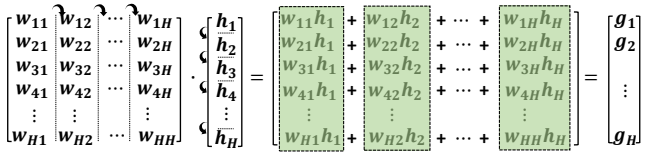


Figure 34: Applying fundamental patterns to $w_h h_{t-1}$.

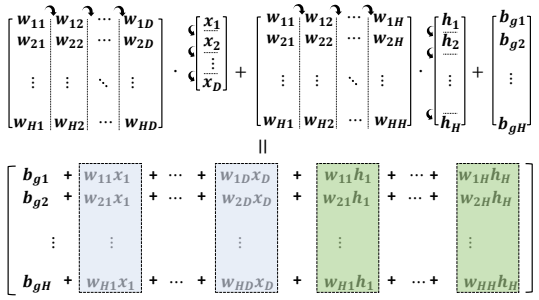


Figure 35: Applying fundamental patterns to a gate.

With the two fundamental computing patterns, the four gates computation can be combined to achieve higher computing parallelism. As operations in element-wise multiplication are independent to each other and x_t, h_{t-1} are shared by the four gates, the weights in the four gates can be stacked vertically to be computed which is shown in Figure 36a. As a

result, the cross-gate kernel computing parallelism can be as large as **4H** as shown in Figure 36b. Therefore, instead of MAC cores and individual kernels for gates, computations in LSTM can be processed in parallel via unified element-wise computing kernel(s). The higher parallelism across gates can also be further taken advantage of parallel-computing devices like FPGA. It is worth mentioning that the two fundamental computing patterns do not bring any extra number of computations.

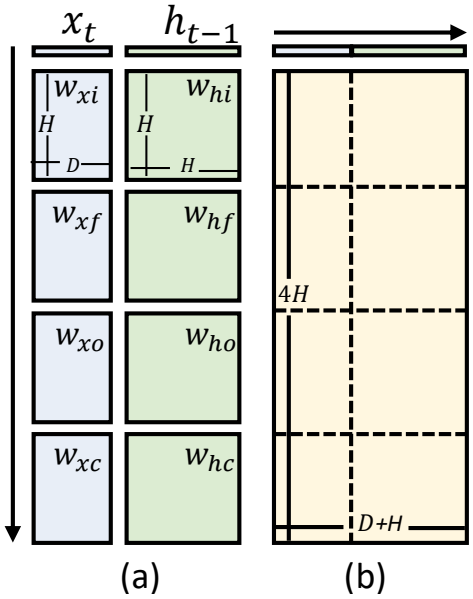


Figure 36: Applying fundamental patterns to four gates.

4.1.3.2 Accelerator based on the Unified Pattern

Thanks to the simplified computing patterns, the accelerator design complexity is reduced, bringing chances to build a unified accelerator kernel for all gates, which benefits accelerator performance in working frequency and resource utilization.

Kernel design: A computing kernel of size $4H$ to process the two patterns (equations 4.6 and 4.7) is shown in Figure 37a. It contains $4H$ pairs of multiplier and adder working in parallel. The multiplier and adder in a pair work in a pipeline fashion. The adder's input is from the multiplier and the previous accumulated result. The computing parallelism in the designed kernel is **4H** and pipeline stages are two. Multiple unified kernels can be instantiated in the accelerator according to device resource constraint.

The accelerator with the kernel of size $4H$ is shown in Figure 37b. Considering the general resources in FPGA platforms, the accelerator consists of on-chip weight buffer, data swapping bus (DMA) between off-chip memory and on-chip buffer, data dispatcher to schedule input data, computing kernel, buffer dispatcher to cache computing results, and routing data bus between internal modules. The weight buffer keeps or partially keeps the LSTM weights during inference, which are off-line transferred to the buffer via data bus ①. During online inference, the buffered weights are read to the computing kernel via data bus ④. The swapping data bus ①, ②, and ③ fetches unbuffered $weight$, x_t, and gate results from off-chip memory during online inference. Data dispatcher routes data such as x_t, buffered $weight$, swapped $weight$, and gate results, sending two vectors into the computing kernel via data bus ⑤ and ⑥. It also includes a ping-pong buffer for online data transfer to hide the transfer time during computing. The computing kernel iteratively computes all the LSTM computations. The buffer dispatcher uses registers to cache kernel output and keep h_t, c_t results on the chip. After finishing off the gate' computations, the buffer dispatcher transfers computation results to the accelerator's master for activations computing in software. Such architecture eliminates the kernel stall and can be easily scaled up or scaled down according to device resource constraint.

Buffer management: The example computing kernel consumes $4H$ weight elements and an element of vector x_t or h_{t-1} simultaneously. Therefore, if the kernel is in size $4H$, a block of buffers in quantities $4H$ is designed to support the kernel. The simultaneously buffer access leads to near-zero reading delay, achieving zero stalls in kernel computing. If the buffer resource is abundant in the device, multiple buffer blocks can be instantiated to support more weight buffering, achieving less system stall caused by data transferring. The buffer depth determines the portion of weight columns that can be buffered on the chip. Thus, a buffer is further partitioned into sections with unit length $D + H$ to hold more weights. The buffer blocks and depth partition will be further discussed in section 4.1.3.3. As the size of x_t and kernel output is usually several magnitudes smaller than weights, the vectors such as x_t, c_t, and h_t are stored in device registers.

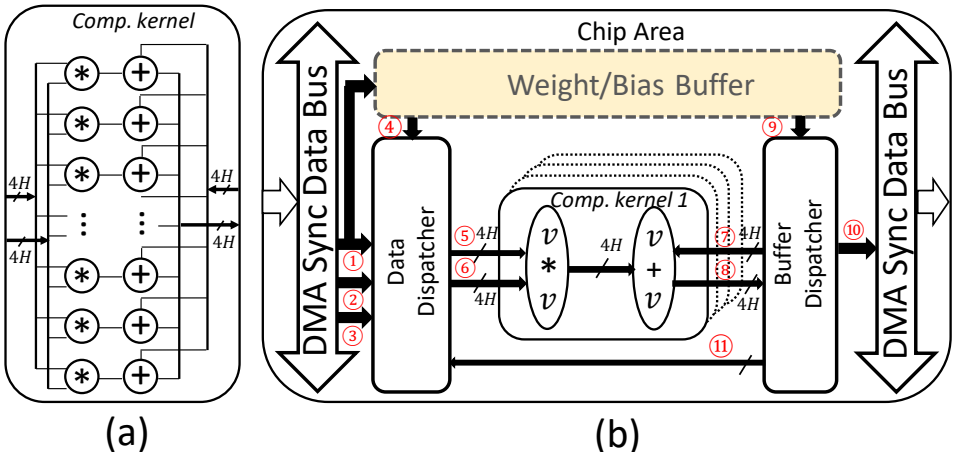


Figure 37: The unified-kernel LSTM accelerator architecture.

4.1.3.3 Analytical Model

Resources in platforms such as FPGA are usually fixed in memory bandwidth, computing units like DSP, logic units like LUT, and on-chip buffer block-RAM (BRAM). Therefore, a device's capacity may not fit an LSTM cell's computing and weights size. The analytical model in this work takes the memory bandwidth (B), the number of DSPs (N_{DSP}), the number of BRAMs (N_{BRAM}), and unit BRAM's depth ($depth$) as the platform parameters. The LSTM's hidden state size H, input x_t vector length D, and inference data bit-width ($width$) are co-considered with the platform parameters.

Hyper-parameters selection. In accelerator parameters selection, the kernel size S_{kernel} should be selected which is less than N_{BRAM}. However, the synthesis strategy varies if FPGA is the target device. For example, Xilinx FPGAs need 5 DSP to compute a single floating-point multiplication and addition in the proposed computing kernel, while 1 DSP is needed for 16 bit fixed-point data type. Therefore, for the target devices, the effective number of DSPs are:

$$N'_{DSP} = \frac{N_{DSP}}{offset} \tag{4.8}$$

The offset represents the DSPs consumed by a pair of multiplication and addition. Then, the upper bound of the kernel size and the number of kernels (N_{kernel}) can be determined:

$$
\begin{aligned}
K_{max} &= min(N'_{DSP}, N_{BRAM}) \\
S_{kernel} &= min(4H, K_{max}) \\
N_{kernel} &= floor(\frac{K_{max}}{S_{kernel}})
\end{aligned}
\tag{4.9}
$$

Considering the target device has N_{BRAM} BRAMs with the unit capacity of *depth* in bit-wise. For a certain application, the BRAM's real unit capacity $depth'$ can be represented by $depth'=\frac{depth}{width}$, where *width* is the application data bit-width. Therefore, the device BRAMs can be modeled as an array with size $[N_{BRAM}][depth']$ as shown in Figure 38. As discussed in section **Buffer Management**, the size of weights to be buffered is $[4H][DH]$, where $4H$ represents $4 * H$ and DH represents $D + H$. Though the physical buffer size and weights size are usually inconsistent, it can be resolved by our following buffer management.

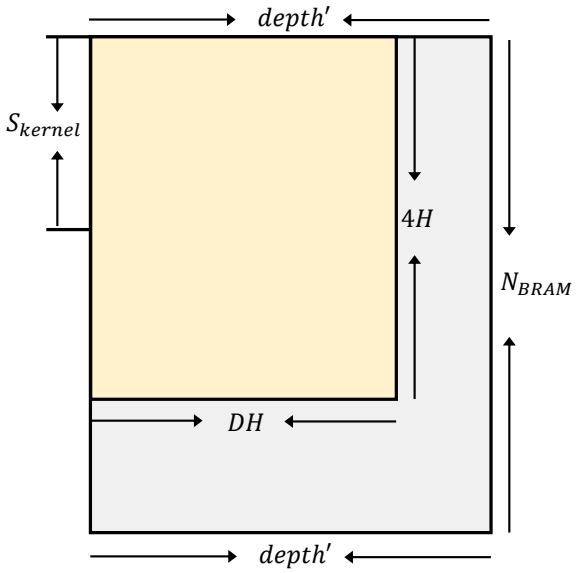


Figure 38: Physical buffer size and weight size.

For the kernel of size S_{kernel}, it reads S_{kernel} buffers simultaneously. Therefore, the BRAMs are first partitioned to $floor(\frac{N_{BRAM}}{S_{kernel}})$ blocks. For the weight matrix with DH columns, each block can hold $\frac{depth'}{DH}$ portion of weight matrix in the column dimension. Herein, the physical buffer capacity (N_{block_p}) of the device for the LSTM is determined by equation 4.10. Meanwhile, the logical buffer blocks (N_{block_l}) needed is determined by the number of weight rows $4H$ and the kernel size S_{kernel} which is shown in equation 4.11.

$$N_{block_p} = floor(\frac{N_{BRAM}}{S_{kernel}}) * \frac{depth^{'}}{DH} \tag{4.10}$$

$$N_{block_l} = ceil(\frac{4H}{S_{kernel}}) \tag{4.11}$$

$$N_{block_t} = min(N_{block_p}, N_{block_l}) \tag{4.12}$$

Therefore, the total weights can be buffered on the chip (N_{block_t}) is determined by equation 4.12. According to N_{block_p}, N_{block_l}, and N_{block_t}, the running schedule can be determined.

Acc. schedule and performance prediction. With the buffered weights, the inference time for processing these weights on the chip is **P1**:

$$t_{\boldsymbol{P1}} = t_{kernel} * N_{block_t} * min(DH, depth^{'})/N_{kernel} \tag{4.13}$$

t_{kernel} is the latency of kernel's single execution. If the physical blocks are less than logical blocks ($N_{block_p} < N_{block_l}$), the non-buffered weighs are transferred from off-chip memory. The inference time under this period is **P2**:

$$t_{\boldsymbol{P2}} = t_{comp} * |N_{block_p} - N_{block_l}| * DH/N_{kernel} \tag{4.14}$$

t_{comp} is the execution time considering the transfer cost, which is $max(t_{kernel}, t_{trans})$ as ping-pong buffer is adopted in the data transferring. t_{trans} is the off-chip weight vector transfer time. The real execution time is determined by t_{kernel} or t_{trans}, the one with the most time cost.

After computing all the computations in the gates, the accelerator takes the gate results from the master and compute c_t and h_t. Similarly, ping-pong buffer is adopted in transfer and inference time is **P3**:

$$t_{\boldsymbol{P3}} = t_{comp} * ceil(\frac{H}{S_{kernel}}) * 3 * t_{comp} \tag{4.15}$$

After **P1**, **P2**, and **P3**, the LSTM cell gate intermediate results are computed. As the activations are done by the master, the gate intermediate results are transferred out of

FPGA. Therefore, the time cost of data transferring and software activation processing time can not be neglected. The cost of data transferring and processing activations is:

$$t_{act} = 5 * H * t_{act\_uni} + \frac{5 * H}{B/width} \tag{4.16}$$

$5 * H$ represents the number of transferred out data for activation and t_{act_uni} represents the unit activation time cost in the master. Therefore, the inference time for the proposed accelerator for LSTM is:

$$t_{LSTM} = t_{\boldsymbol{P1}} + t_{\boldsymbol{P2}} + t_{\boldsymbol{P3}} + t_{act} \tag{4.17}$$

According to metrics 4.8 to 4.17 in the analytical model, the accelerator parameters, accelerator running schedule, and accelerator performance can be acquired.

4.1.4 Experiment

4.1.4.1 Experiment Setup

The LSTM FPGA accelerator in both 16-bit fixed-point (named as Uni.16) and 32-bit floating-point data (named as Uni.32) are built respectively. The accelerator is implemented on Xilinx PYNQ-Z1 SoC FPGA with a Cortex-A9 ARM processor and FPGA chip XC7Z020-1CLG400C. The accelerator is designed in Xilinx Vivado HLS (v2018.3) and synthesized by Vivado (v2018.3). The ARM processor works as the master for scheduling and activation; the FPGA accelerator works as the slave for computing.

4.1.4.2 Performance Evaluation

We show the performance of our design in inference latency, accelerator throughput, resource utilization, and power efficiency in processing a real-life LSTM model. The LSTM has the hidden state size 128 (H) and input size 65 (D). The number of computations processed by our accelerator in this model is 0.1984M (512*193*2+128*3*2).

The latency performance comparison of our accelerator with an existing accelerator (Exist.16 [67]) on FPGA in 16-bit fixed-point data is shown in Table 4. The reference design

Table 4: Performance comparison for designs on 7Z020

	Exist.16[67]	**Our.16**
Chip	7Z020	7Z020
Frequency	142MHz	150MHz
Precision	fixed-16	fixed-16
Operations	0.1332M	0.1984M
Latency	466μs	**46.7**μs
Throughput	0.28GOP/s	**4.25GOP/s**

adopts the heterogeneous-kernel architecture to process different computing patterns. In this table, chip name, working frequency, data type, number of operations, inference latency, and the throughput performance are listed. Towards accelerating the same LSTM on the same device, the existing design needs 466μs while our design takes 46.7μs to finish the inference. The latency of our design is 10x faster than the reference. The accelerator throughput performance is shown in Giga Operations per second (GOP/s). Our accelerator achieves 4.25GOP/s, which is 15.2x higher than the reference design.

To show the superiority of our work, we list our accelerator performance in fixed-point and floating-point types (Uni.16, Uni.32) in Table 5, while the reference design does not support floating-point computing. The floating-point accelerator's inference latency is 458μs because floating-point computation needs more resources and a longer time. For the throughput performance, the floating-point accelerator achieves 0.43GFLOP/s, where FLOP represents the floating-point operation. Our works in both fixed-point and floating-point data outperform the reference design.

The resource utilization of the existing designs and our accelerators are shown in Table 6. In this table, the resource of computing units (DSP), on-chip buffer (BRAM), flip-flops (FF), and LUT are listed. As shown in the table, the existing design only utilizes 23% of the DSP and 11% of the BRAM, while our accelerator achieves 82% and 73% DSP utilization; 64% and 92% BRAM utilization in fixed and floating type respectively. The higher DSP utilization brings more computing power and the higher BRAM utilization brings less weight transfer

Table 5: Performance details of the proposed design

	Our.16	**Our.32**
FPGA chip	7Z020	7Z020
Frequency	150MHz	150MHz
Precision	fixed-16	float-32
Operations	0.1984M	0.1984M
Latency	$46.7\mu s$	$458\mu s$
Throughput	4.25GOP/s	0.43GOP/s

time. The DSP usage is limited by the LUT resource on FPGA, which is the auxiliary logic resource. We have 3.6x and 3.2x improvement in DSP utilization. Among the utilized DSPs, all the DSPs in our accelerator can work in parallel, achieving zero hardware idle time. The DSPs in the existing design can only be partially invoked[67]. Therefore, our accelerator architecture achieves super-linear inference time speedup in **10x** faster while using **3.6x** more DSPs.

Table 6: Resource comparison for designs on 7Z020

Resource	**DSP**	**BRAM**	**FF**	**LUT**
Total	220	280	106400	53200
Exist.16[67]	23%	11%	12%	14%
Our.16	**82%(x3.6)**	**64%**	65%	97%
Our.32	**73%(x3.2)**	**92%**	58%	100%

Table 7: Power comparison for designs on 7Z020

	Exist.16[67]	**Our.16**	**Our.32**
Power	$1.95W$	$2.29W$	$2.23W$
Efficiency	0.146GOP/s/w	1.86GOP/s/w	0.193GFLOP/s/w

As a result of more resource utilization and less running stall, the proposed accelerator has slightly higher power consumption which is shown in Table 7. The reference design, our fixed, floating-point accelerators (Exist.16, Uni.16, Uni.32) have working power $1.95W$, $2.29W$,

and 2.23W. However, our accelerator is much more power efficiency in Giga Operations per second per watt (GOP/s/w). Our fixed-point accelerator achieves 1.86GOP/s/w, which gains 12.7x power efficiency than the reference design. Our floating-point accelerator achieves 0.193GFLOP/s/w. The power measurement via power meter for fixed and floating-point accelerators is shown in Figure 39.

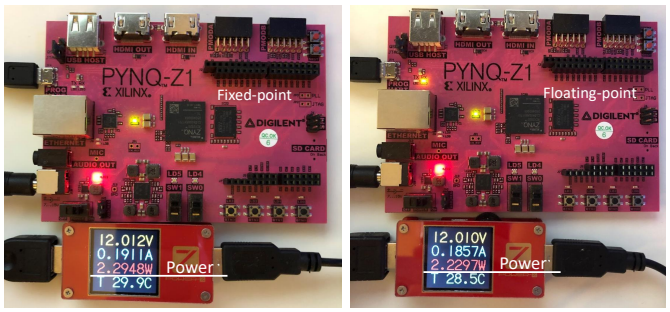


Figure 39: Power measurement of accelerators on 7Z020.

4.1.4.3 Analytical Model Accuracy Analysis

The hyper parameters of the accelerator are shown in Table 8. According to metric 4.8 and 4.9, the number of computing kernel is 1 for both data types; the upper bound of kernel size is initialized as 220 and 44 for fixed-point and floating-point data respectively. When compiling the design template with initialized kernel size, the resource utilization feedback can be acquired from the tool (HLS) in a few seconds. With the tool, the kernel size for fixed-point and floating-point accelerators are selected as 180 and 32 under the constraint of LUT resource. After kernel size selection, according to metrics 4.10, 4.11, and 4.12, the device buffer capacity N_{block_p} and logical blocks N_{block_l} are 9.1 and 3 for fixed-point accelerator; 25.5 and 16 for floating-point accelerator. Therefore, both the fix-point and floating-point accelerators can buffer all the weights on the chip.

Based on the evaluations above, both accelerators will undergo period **P1** and **P3**. For FPGA chip 7Z020, it takes 5 cycles for one-time kernel execution (t_{kernel}) for fixed-point and 18 cycles for the floating-point data. The unit activation time in ARM (650MHz) is equivalent to 3.9 FPGA cycles (150MHz). Via metrics 4.13, 4.15, 4.16, and 4.17, we get the predicted accelerators' inference latency in cycles which is shown in Table 9 where the

on-board latency is also listed. The analytical model we proposed achieves a performance prediction deviation of 6.2% in fixed-point accelerator and 2.7% in floating-point accelerator. As **P2** and **P3** share parameters t_{kernel}, t_{trans}, and t_{comp}, the accuracy of **P2** is also verified. The low deviation proves that our analytical model can accurately determine accelerator hyper parameters, arrange the running schedule, and predict the system performance.

Table 8: Accelerators hyperparameters for 7Z020

	N_{kernel}	S_{kernel}	N_{block_p}	N_{block_l}	N_{block_t}
Uni.16	1	180	9.1	3	3
Uni.32	1	32	25.5	16	16

Table 9: LSTM accelerator performance model accuracy analysis.

Design	Our Model			On-Board			Deviation		
	Cycles	BRAM	DSPs	Cycles	BRAM	DSPs	Cycles	BRAM	DSPs
Our.16	7864	220	220	8383	180	180	6.2%	-	-
Our.32	69392	264	220	70296	258	160	2.7%	-	-

4.2 Software/Hardware Co-design for Transformer Network

4.2.1 Background

Transformer [47] is an emerging DNN model that achieves competitive and even better performance than LSTM. It has been widely applied in popular DNN models such as BERT [78] in language modeling, GPT [79] in the general language model, and DETR [80] in image processing. The size of the Transformer-based model ranges from million bytes (Transformer) to billion (GPT-3) bytes.

While large-scale Transformer-based DNN models are developed to break the records in the Natural Language Processing (NLP) tasks, the computations become more and more intensive. As the majority of the computations in the self-attention mechanism are matrix

multiplications, it has been reported that 10 Giga multiply-accumulate operations (MAC) are needed when translating a short sentence via Transformer [81]. With such a large number of weights and MACs, a large memory footprint and high computational cost are demanded when deploying the mechanism.

Existing works proposed to structurally prune the model weight to keep computing efficiency. The memory footprint and workload are largely reduced after pruning. Such method focuses on removing the redundant weights without hurting the robustness of the model architecture, which also avoids the efforts in proposing new models. The efficient pruning method for attention mechanism has been observed in TransformerZip [82], HAT [83], and FTRANS [21]. TransformerZip [82] utilizes magnitude-based pruning to reduce weight size; HAT [83] crops the weights in both dimensions to reduce weight size and form regularly shaped weights. FTRANS [21] utilized block-circulant matrix to replace selected weights, which reduces the model memory footprint.

When programming and deploying weight-pruned models in the inference stage on generic processors, e.g., CPUs and GPUs, it incurs little effort. However, it poses significant programming efforts and design challenges on hardware accelerators like ASICs and FPGAs for the following two reasons: First, the dimension size of different weights can be arbitrary after compression as the significance or absolute value differs among the weight elements. It is challenging to efficiently allocate on-chip buffers for different shapes of weight under on-chip hardware constraints to maximize buffer utilization and improve inference throughput. Second, the accelerator is usually computing pattern-specific. While the compression causes arbitrary-sized weight and its associate computing pattern, building a dedicated computing kernel for each pattern is not feasible. As buffer allocation and computing kernel design on the hardware accelerator is usually fixed in size and limited in number for specific computations, these two issues may severely hurt the accelerator efficiency. Therefore, the Transformer's memory footprint and hardware computing pattern should be jointly considered and optimized when deploying the Transformer on the hardware accelerator.

4.2.2 Motivation and Contribution

The existing works proposed efficient methods to address the large memory footprint size of weights. However, the weight compression and computation are not jointly considered, which leads to inefficient utilization of on-chip memory and compute resources, resulting in computing inefficiency. Figure 40 (a) and (b) show the compression result of one self-attention in [83]. The intermediate results related to the pruned weights are also white-colored for better illustration. [83] crops the columns of Q^w, K^w, and V^w. However, the width of compressed weight is not controlled, which also directly influences the MAC size of $Q * K^T$as shown in Figure 40 (b). The linear layer weights in FFN are even arbitrary in both height and width which are shown in Figure 40 (c). As an attention mechanism consists of 8 heads and the Transformer consists of 6 encoders and 6 decoders, the number shapes of the compressed weights and the associated computation size can be as high as several hundred. Such compression method causes severe buffer inefficient usage on the hardware accelerator. Since MAC is usually executed in parallel on the accelerator, the suitable computing kernel size also varies for those compressed weights. As a result, the solution adopted in [21] will cause extreme unbalance computing kernel design and difficult pipeline arrangement. The accelerator system performance may be largely degraded.

In this work, targeting on Transformer in machine translation task, we propose to compress the Transformer model with weight size awareness, leaving weights in similar size. The similarity of the compressed weights brings more efficient hardware buffer utilization. A novel computing pattern is also proposed to address the computing heterogeneity and inefficiency in the Transformer after compression. With the compression methodology and the unified computing pattern, an FPGA accelerator is designed to accelerate the sparse matrix multiplications of Transformer. The accelerator is recursively called in a mode of streaming in, computing, and streaming out. In this way, the accelerator with the fixed buffer and the unified computing core is efficient to handle the deployment of sparse Transformer.

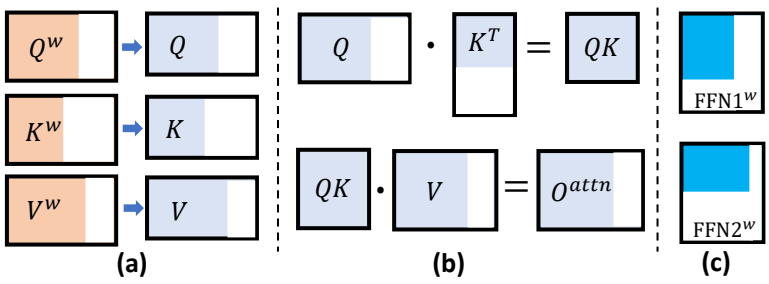


Figure 40: Related works.

4.2.3 Algorithm Optimization

The algorithm optimization is performed according to the means of "Winning Ticket Hypothesis' [84]. 'Winning ticket' theory proves that the sub-network of a model can have comparable accuracy by correctly removing the smallest-magnitude weights and training from original initialization (one-shot pruning [84]). The 'winning ticket' theory has been adopted and validated in the existing works for CNN models [84, 85, 86, 87], in which, [85] identifies Normalization is efficient in identifying the Convolution weight magnitude channel-wise. However, Transformer has a different architecture with CNN models, the existing techniques can not be directly applied to Transformer. In this work, we propose a novel 'winning ticket' finder, in which, layer normalization (LayerNorm) [88] is adopted to firstly analyze the weight significance in column-wise.

4.2.3.1 Weight Significance Analysis before Model Compression

In order to analyze the weight significance, normalization modules are adopted to analyze the weight significance in column-wise. As normalization has been proved effective to reflect the weight channel-significance in the Convolution [84, 85], in this work, layer normalization (LayerNorm) [88] is adopted to identify the weight significance in column-wise. The developed analysis workflow is shown in Algorithm 1.

LayerNorm is exclusively applied to one encoder or one decoder layer at a time (denoted as *model.layer* in the algorithm), in which, LayerNorm is attached to any matrix multiplication that contains weight (*module.weight* in the algorithm) in the layer. The Transformer is

Algorithm 1: Weight significance analysis

Input: dataset ($data$), Transformer ($model$),

Transformer total layer L, $LayerNorm$.

Output: $model_{norm}$.

$model_{norm} = model$;

for $l \leftarrow 1$ **to** L **do**

  $model_l = model$;

  for $module \in model_l.layer[l]$ **do**

    if $module.weight$ **then**

      $module.attach(LayerNorm)$;

    end

  end

  while $LayerNorm^* \in model_l$ ***Not*** *Converged* **do**

    $model_l.train$;

  end

  $model_{norm}.layer[l] = model_l.layer[l]$;

end

then trained until the scaling factors γ in the newly attached LayerNorms are converged. The converged γ is collected as the column significance indicator for the weight. After repeatedly applying LayerNorms to each encoder and decoder layer and training the Transformer, the scaling factors γ in all of the attached LayerNorms are collected and stored in a separated model $model_{norm}$. The visualization of an encoder before and after LayerNorm attachment is shown in Figure 41 (a) and (b), in which, the newly attached LayerNorm is labeled as $LayerNorm^*$.

The LayerNorm takes a vector or a matrix as input and scales the row elements individually which is shown in Equation (4.18). In a LayerNorm, the row elements expectation E and variation Var are employed; the scaling factors γ and β are dedicated for each row element but shared among rows. As visualized in Figure 42, a pair of the γ and β can scale a column of the inputs up or down. When attaching the LayerNorm to matrix multiplication, a column of the results will be normalized by the same γ and β. Therefore, a pair of the scaling factor γ and β can reflect the significance of the corresponding column of the weight. As γ dominates the scaling operation, we take γ value as a significance indicator. The scaling degree can be reflected by γ value. The weight significance can be ranked via its corresponding γ while weight with larger γ is more important. According to the significance ranking, the columns of the weights with smaller γ will be pruned. In the next step, the method to prune the weight in columns-wise with memory footprint awareness will be illustrated.

$$x_{scaled} = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta \tag{4.18}$$

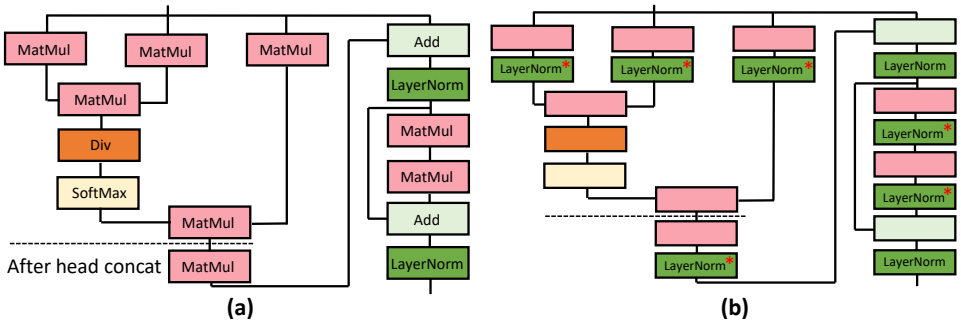


Figure 41: LayerNorm insertion of the encoder.

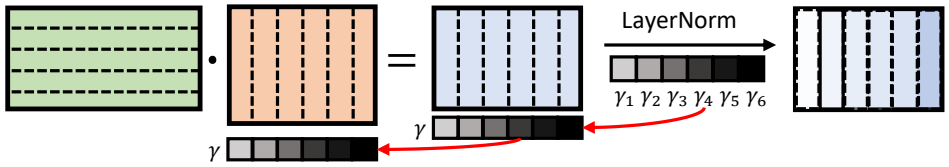


Figure 42: LayerNorm scaling factor visualization.

4.2.3.2 Pruning Strategy

To ensure the hardware efficiency after pruning, a two-stage one-shot pruning is performed: coarse-grained pruning to keep the weights of the same type in similar size; fine-tune to adequately remove the redundant weights without losing accuracy. In each stage, pruning and training are pairwise performed until the model is adequately pruned within the stage.

Coarse-grained. the coarse-grained pruning prunes each of the weights in the Transformer with the same ratio. During this stage, the memory footprint size of the weights is evenly reduced. Based on the dataset *data*, the Transformer *model*, the collected γ data in $model_{norm}$, and a pre-set pruning speed $incr_{coarse}$, coarse-grained pruning will determine the maximum even pruning ratio. As shown in Algorithm 2, the baseline accuracy is acquired first. Then, the pruning ratio increases by $incr_{coarse}$ from 0%. At each pruning ratio, **GetIndex** will locate the index of γ ranging in the least $ratio$% in a LayerNorm. Such index is equivalent to the index of weight columns. According to the selected index, **Mask** will mask the indexed weight columns to enable zero gradient descent during training. The increment of $ratio$ stops when the sparse model accuracy drops below the baseline accuracy ($accuracy_{base}$). After this stage, the maximum size of each weight is bounded.

Fine-tune. In this stage, the masked columns in stage one are considered as already been removed in both *model* and $model_{norm}$. As shown in Algorithm 3, fine-tune takes *data*, coarsely pruned *model*, $accuracy_{base}$, $model_{norm}$, and $incr_{fine}$ as input, performing across layer pruning based on the rest γ. In fine-tune, encoders and decoders are pruned separately. Algorithm 3 takes encoders as an example since encoder and decoder are similar in architecture. The γ of LayerNorms in $model_{norm}$ for the same type weights cross all encoder layers are grouped first. For example, the γ for Q^w ($layer.module.LayerNorm$ in the algorithm) in

Algorithm 2: Coarse-grained pruning

Input: $data$, $model$, $model_{norm}$, $incr_{coarse}$.

Output: Pruned $model$, $accuracy_{base}$, $ratio_{coarse}$.

$model.train$;

$accuracy_{base} = model.accuracy$;

$ratio = 0\%$;

while $True$ **do**

  $ratio = ratio + incr_{coarse}$;

  for *module in Encoders* **do**

    if $module.weight$ **then**

      $GetIndex(Norm_{layer}.module.LayerNorm, ratio)$;

      $Mask(model.layer.module.weight)$;

    end

  end

  $model.train$;

  if $model.accuracy < accuracy_{base}$ **then**

    $ratio_{coarse} = ratio - incr_{coarse}$;

    break;

  end

end

/* GetIndex: locating the index of γ ranging in the least $ratio$ percent. */

/* Mask: masking the selected column to ensure zero gradient descent in training. */

Algorithm 3: Fine-tune pruning

Input: *data*, coarsely pruned *model*, $accuracy_{base}$, $model_{norm}$, $incr_{fine}$.

Output: Fine pruned *model*.

$EncQ_{norm}$; $EncK_{norm}$; $EncV_{norm}$; $EncO_{norm}$; $EncFn1_{norm}$; $EncFn2_{norm}$;

for *layer in* $model_{norm}.EncoderLayers$ **do**

  for $module \in layer$ **do**

    if *module.weight* **then**

      switch *module.Name* **do**

        case *Q* **do**

          $EncQ_{norm}.append\ (layer.module.LayerNorm)$;

        case *K* **do**

          $EncK_{norm}.append\ (layer.module.LayerNorm)$;

        case *V* **do**

          $EncV_{norm}.append\ (layer.module.LayerNorm)$;

        case *O* **do**

          $EncO_{norm}.append\ (layer.module.LayerNorm)$;

        case *Fn1* **do**

          $EncFn1_{norm}.append\ (layer.module.LayerNorm)$;

        case *Fn2* **do**

          $EncFn2_{norm}.append\ (layer.module.LayerNorm)$;

      end

    end

  end

end

$ratio = 0\%$;

while $True$ **do**

  $ratio = ratio + incr_{fine}$;

  $GetIndex(EncQ_{norm}, ratio)$; $Mask(model.EncQ.weight)$;
  $GetIndex(EncK_{norm}, ratio)$; $Mask(model.EncK.weight)$;
  $GetIndex(EncV_{norm}, ratio)$; $Mask(model.EncV.weight)$;
  $GetIndex(EncO_{norm}, ratio)$; $Mask(model.EncO.weight)$;
  $GetIndex(EncFn1_{norm}, ratio)$; $Mask(model.EncFn1.weight)$;
  $GetIndex(EncFn2_{norm}, ratio)$; $Mask(model.EncFn2.weight)$;

  $model.train$;

  if $model.accuracy < accuracy$ **then**

    $ratio = ratio - incr_{fine}$;

    break;

  end

end

```
/* GetIndex:  locating the index of γ ranging in the least ratio% in the group.                  */
/* Mask:  masking the selected column in the group to ensure zero gradient descent in training.  */
```

six encoder layers are grouped. After collecting the cross-layer γ for each type of weight, γ for corresponding weights are stored in $EncQ_{norm}, EncK_{norm}, EncV_{norm}, EncO_{norm}, EncFn1_{norm}$, and $EncFn2_{norm}$. The pruning and training will be executed similarly as in Algorithm 2. However, in this algorithm, **GetIndex** will locate the index of rest γ ranging in the least ratio% in the group. For example, if the *model* consists of six encoders, the γ in all of the encoders are compared. **Mask** also performs cross-layer masking in this stage. After this stage, the Transformer model size is further reduced with slight size variation.

The two-stage pruning adequately prunes the model without accuracy loss. The pruned weights in Transformer are illustrated in Figure 43 and Figure 44, in which, the pruned weights and corresponding results are white-colored for better illustration. As a result, the weights in same type (e.g., $Q^w_{layer}, K^w_{layer}, V^w_{layer}, O^w, FFN1^w$, and $FFN2^w$) in the Transformer will be in similar size. This will maximize the utilization efficiency of the data buffer on hardware. The well-trained sparse *model* can be directly quantized to INT8 data via dynamic quantization with Pytorch [89] without accuracy loss. After quantization, the model size can be further reduced by 4x, which benefits a smaller memory footprint and more efficient hardware computing.

4.2.4 Hardware Optimization

4.2.4.1 Unified Computing Pattern in Sparse Transformer

With the observation that compression forms fixed height (N) for multiplicand and multiplier of the matrix multiplications in the model, the MAC of different sizes can be replaced by a sequence of uniform element-wise vector multiplication and addition. The

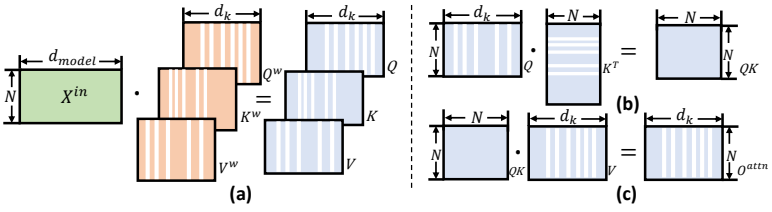


Figure 43: Sparse self-attention computations.

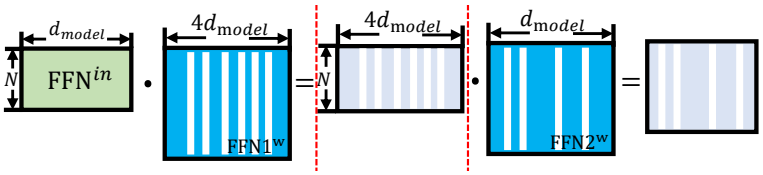


Figure 44: Sparse FFN computations.

unified computing pattern is shown in Fig. 45, in which IN/WEI represents the multiplicand and multiplier respectively and OUT represents the product. OUT is computed column by column. Each column of OUT is partially computed by multiplying an element in Y and its corresponding column of IN, the accumulation of column one is shown in this figure (the element is duplicated to a vector). The pseudocode of this process is also shown in Figure 46, where the inner loop computes the partial result of a column in OUT and can execute in parallel on FPGA. After the iterations of the middle loop, a column of Z is computed. As highlighted in the loops, the iterations of the inner loop are the height of the multiplicand which is fixed (N) cross all the matrix multiplications in Transformer; the $WEI[row_{wei}][col_{wei}]$ is a constant in the inner loop iterations. The efficiency of such unified computing pattern is validated by my previous work [65].

Such computing pattern can further enlarge the FPGA DSP efficiency. DSP components in modern FPGAs are usually designed with high bitwidths such as $27b * 18b \rightarrow 45b$ [90], which is redundant for an INT8 operation. The DSP can only be fully utilized by encoding INT8 multiplication as $((a <<) + b) * c$ [90] which is shown in Fig. 47. However, the shared multiplier (c) for multiplicands (a and b) do not exist in the conventional MAC. As the element $WEI[row_{wei}][col_{wei}]$ is shared in the inner loop, the DSP bitwidth can be fully utilized by encoding INT8 as $((a <<) + b) * WEI[row_{wei}][col_{wei}]$. The followed addition can be executed in FPGA LUT.

4.2.4.2 Accelerator Design

Processing element. The diverse computations are unified into patterns: element-wise vector multiplication and addition, which can be conducted by only two types of Processing Elements (PE). The element-wise vector multiplication between a multiplicand's column

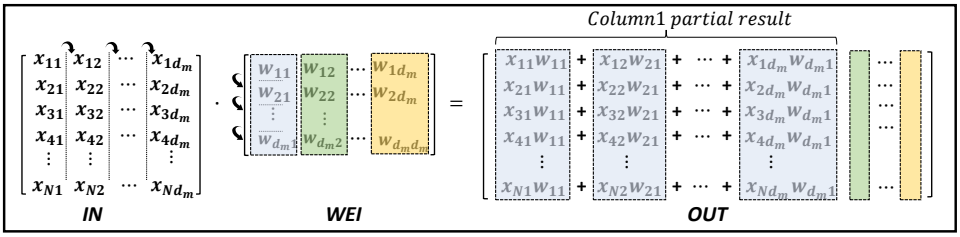


Figure 45: Unified computing pattern.

```
for (col_wei=0; col_wei< C_WEI; col_wei++) {
   for (row_wei=0; row_wei< R_WEI; row_wei++) {
      for (row_in=0; row_in< R_IN; row_in++)      {
         OUT[row_in][col_in] +=IN[row_in][row_wei]*WEI[row_wei][col_wei]
} } }
/*  R represents the number of rows  */
/*  C represents the number of columns */
```

Figure 46: Loop iteration of the unified computing pattern.

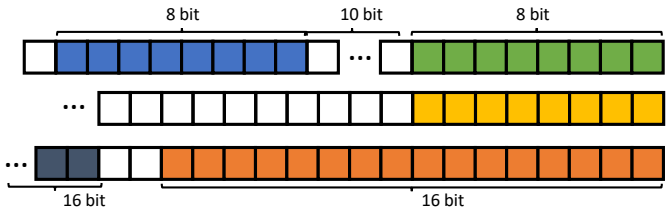


Figure 47: INT8 multiplication encoding.

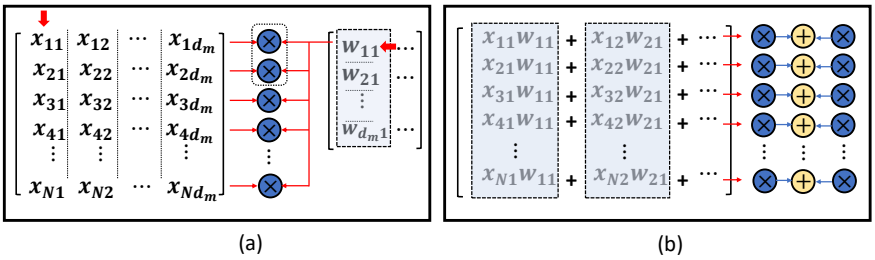


Figure 48: (a) The PE1 architecture and PE1 mapping to the multiplication in the unified computing pattern. (b) The PE2 architecture and PE2 mapping to the addition in the unified computing pattern.

and an element of the multiplier can be executed via multiplication unit $PE1$, in which, the parallelism of N ($N = R_{IN}$) can be performed (illustrated by the inner loop of Figure 46). The architecture of $PE1$ and $PE1$ to IN and WEI access is shown in Figure 48 (a). In $PE1$, the multiplications are mainly implemented via FPGA DSPs and every two of the multiplications are packed into one DSP according to the INT8 data packing mechanism illustrated in section 4.2.4.1. The partial results regarding all multiplicand's columns can be accumulated via addition unit $PE2$ in the parallelism of N as shown in Figure 48 (b). $PE2$ is implemented via FPGA LUT as INT8 addition is more efficient with LUT resources. More $PE1$ can be built with FPGA LUT resource as long as the LUT resource is abundant. By building and connecting multiple $PE1$ and $PE2$, the unified computations can be efficiently processed.

Computing core. As the height of the multiplicand of matrix multiplications in Transformer is fixed at N, a homogeneous $PE1$ array and homogeneous $PE2$ array are utilized to build a multi-stage pipeline computing core. The PE array processes the unified computations in high parallelism. The multiplicand's columns can be processed simultaneously via $PE1$ array and accumulated by the following tree-structured $PE2$ array. At the lowest hierarchy of $PE2$ tree, an additional $PE2$ accumulates the partial results. A 6-stage computing core with 8 $PE1$ and 8 $PE2$ is shown in Figure 49 (a), in which, the parallelism of $PE1$ and $PE2$ is set as N. 8 $PE1$ work in parallel and produce 8 vectors in length N. Each $PE2$ in the "adder-tree" structure accumulates two vectors. Buffers are placed between the

hierarchies to support pipeline execution. The accumulation for all $PE1$ results is acquired after the data flow reaches the bottom $PE2$. As a result, after $R_{WEI}/8$ executions of the computing core, the first column of the output is acquired.

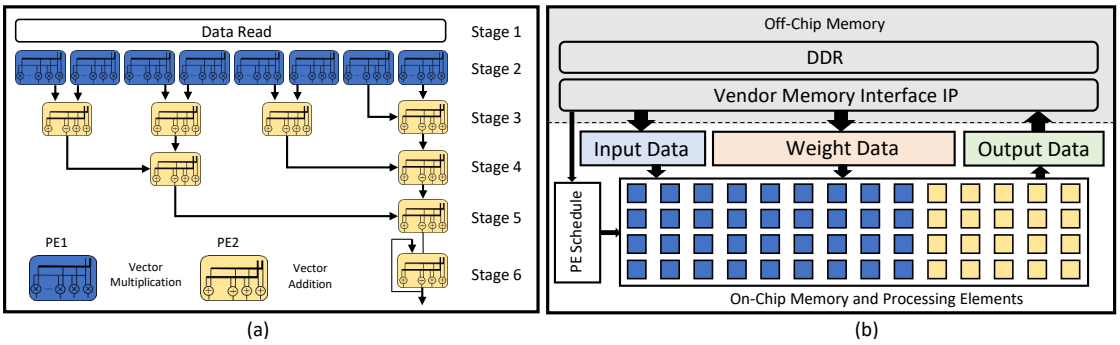


Figure 49: (a) Computing core hierarchy. (b) Accelerator architecture overview.

Since the size of the computing core is bounded by the device resource, a generic data flow of the proposed computing core is shown in Figure 50. The pipeline stage $M + 1$ is determined by the number of $PE1$, where the M stages are spent on computation within PE array and the 1 stage is spent on buffer access. A larger $PE1$ array only leads to a slightly deeper pipeline since each hierarchy in "adder-tree" halves the output of $PE1$. The computing core enter $INITIAL$ to read input and weight elements in the buffer. In $STAGE\_1$, $PE1$ array multiplies the corresponding input columns with weight elements. In the rest $STAGE$, $PE2$ at each hierarchy reads the intermediate results from its higher-level hierarchy and does the accumulation. Such design can maximize the multiplication parallelism and pipeline efficiency. By halving the partial results at each $STAGE$, only $log(2)N_{PE1} + 2$ pipeline stages are needed for the computing core. The pipeline interval of the proposed computing core architecture can run at II=1 during on-board execution.

Data Fetch	Input, weight	Input, weight	STAGE_1 output	STAGE_2 output	...	STAGE_M-1 output
Computation	N/A	Multiply (PE1)	Add (PE2)	Add (PE2)	...	Add (PE2)
STAGE	INITIAL	STAGE_1	STAGE_2	STAGE_3	...	STAGE_M

Figure 50: The data flow of the computing core.

Accelerator system design. While the memory footprint and the workload are largely reduced by the proposed compression method, the compressed Transformer may still exceed FPGA capacity. Therefore, an accelerator system design that includes modules such as data

swapping, data computing, and running scheduler is needed. The proposed full system design is shown in Figure 49 (b) in consideration of off-chip memory (DDR), an input data buffer (buf_{in}), a weight data buffer (buf_{wei}), an output data buffer (buf_{out}), PE schedule register, and the computing core. Each buffer utilizes FPGA on-chip block-RAM (BRAM) and LUT-RAM to store the two-dimension data. At each on-chip buffer to DDR interface, a dedicated streaming bus with ping-pong buffer is placed to support continuous processing. The PE schedule register stores the data fetch information and the number of executions (R_{WEI}/N_{PE1}) to get a single column of output and the whole output columns (C_{WEI}). Per computing core execution, N_{PE1} partial results are accumulated. After R_{WEI}/N_{PE1} executions, a column of output is acquired. Therefore, after $(R_{WEI}/N_{PE1}) * C_{WEI}$ execution, the full output elements are acquired. By reading the execution information from PE schedule register, the corresponding address of the input buffer, and the associated elements of weight buffer, the computing core generates the full elements of the output. In this way, the proposed accelerator can be recursively called to process the matrix multiplications in Transformer while minimizing the system stall caused by transmission.

Accelerator buffer allocation. The size of three buffers can be determined by the compressed Transformer size and the computing core. Among the six types of weights, Q^w_{layer}, K^w_{layer}, V^w_{layer}, and O^w are in the same size as $N_{head} * d_k = d_{model}$ in Transformer. And the size $FFN1^w$ and $FFN2^w$ is quadruple of Q^w_{layer}, K^w_{layer}, V^w_{layer}, and O^w. Therefore, before compression, by allocating buf_{wei} of size $R^{d_{model} \times d_{model}}$, the accelerator can recursively process the computations related to all these weights. During compression, the maximum size of Q^w_{layer}, K^w_{layer}, V^w_{layer}, O^w, $FF1^w$, and $FF2^w$ is determined by $ratio_{coarse}$ in the coarse-grained pruning stage. Though fine-tune pruning unevenly compresses the weights, after compression, the buf_{wei} of size $R^{d_{model} \times (d_{model} * (1 - ratio_{coarse}))}$ is able to process different weights. After determining the size of buf_{wei}, the size of buf_{in} and buf_{out} can be determined. The first dimension of buf_{in} is N since height of the multiplicand of matrix multiplications in Transformer is fixed at N. The second dimension of buf_{in} is d_{model} which is the same as buf_{wei}'s first dimension. The first dimension of buf_{out} is N, which is consistent to buf_{in}. The second dimension of buf_{out} is $d_{model} * (1 - ratio_{coarse})$ which is the same as buf_{wei}'s second dimension. The buffer allocation summary is shown in Equation (4.19).

In this way, the matrix multiplications in Transformer such as $X^{in} * Q^{w}_{layer}$, $X^{in} * K^{w}_{layer}$, $X^{in} * V^{w}_{layer}$, $Q * K^{T}$, $QK * V$, $O^{attn}_{layer} * O^{w}$, $x^{FFN1} * FFN1^{w}$, and $x^{FFN2} * FFN2^{w}$ can be processed via a pipeline processing pattern of streaming in, computing, and streaming out. When the on-chip buffer size is abundant, the capacity of buf_{wei} can be expanded in its second dimension. While the original part $d_{model} * (1 - ratio_{coarse})$ is still used for weight streaming, the expansion part of buf_{wei} can be used to cache selected weights on the chip to reduce off-chip transmission overhead.

$$
\begin{aligned}
buf_{wei}[R_{WEI}][C_{WEI}] &= buf_{wei}[d_{model}][d_{model} * (1 - ratio_{coarse})] \\
buf_{in}[R_{IN}][C_{IN}] &= buf_{in}[N][d_{model}] \\
buf_{out}[R_{OUT}][C_{OUT}] &= buf_{out}[N][d_{model} * (1 - ratio_{coarse})]
\end{aligned}
\tag{4.19}
$$

Streaming interface design. As the Transformer's input and part of the weights are stored in the off-chip memory, the accelerator buffer design and transmission bandwidth are also optimized to minimize the transmission latency. Ping-pong buffer is applied on buf_{wei}, buf_{in}, and buf_{out}, forming buf_{wei}, buf_{weiDB}, buf_{in}, buf_{inDB}, buf_{out}, and buf_{outDB} at the interface. The off-chip memory transmission latency is hidden by alternately accessing the ping-pong buffers in two modes. Mode 1: off-chip to buf_{wei} and buf_{in} transmission, and buf_{out} to off-chip transmission while the PE array reads/writes data from/to buf_{weiDB}, buf_{inDB}, and buf_{outDB}. Mode 2: off-chip to buf_{weiDB} and buf_{inDB} transmission, and buf_{outDB} to off-chip transmission while the PE array reads/writes data from/to buf_{wei}, buf_{in}, and buf_{out}. In this way, the PE array access the ping-pong buffers alternately, hiding the off-chip transmission latency. In addition to the ping-pong buffer, the streaming interface bandwidth is also partitioned according to the buffer size. The streaming bandwidth for buf_{wei}, buf_{in}, and buf_{out} are assigned according to the ratio of a buffer capacity to total buffer capacity (e.g. streaming bandwidth for buf_{wei}: $B_{wei} = B_{total} * \frac{Capacity_{buf_{wei}}}{Capacity_{buf_{wei}} + Capacity_{buf_{in}} + Capacity_{buf_{out}}}$).

Accelerator buffer partition. The input buffer (buf_{in}), weight buffer (buf_{wei}), and the output buffer (buf_{out}) store two-dimension data via BRAM and LUTRAM. The buffers should also be carefully partitioned to support the parallel access from the computing cores.

For input buffer $\boldsymbol{buf_{in}}$, since $PE1$ array accesses multiple columns of input buffer simultaneously, the $buf_{in}[R_{IN}][C_{IN}]$ should be partitioned by parameter N_{PE1} in its second dimension. As each $PE1$ access all the elements in a column of buf_{in}, the buf_{in} should be fully partitioned in its first dimension. For weight buffer $\boldsymbol{buf_{wei}}$, since a $PE1$ access one element in $buf_{wei}[R_{WEI}][C_{WEI}]$ and the $PE1$ array access N_{PE1} elements in buf_{wei} simultaneously, buf_{wei} should be partitioned by parameter N_{PE1} in its first dimension. For output buffer $\boldsymbol{buf_{out}}$, as the last $PE2$ write its elements to the first dimension $buf_{out}[R_{OUT}][C_{OUT}]$ simultaneously, buf_{out} should be fully partitioned in its first dimension.

Accelerator running schedule. With the allocated three buffers and the computing core, the accelerator can continuously process the matrix multiplications (aka Linear modules). The running schedule of the accelerator is shown in Figure 51. With the adopted ping-pong buffer, the streaming of buf_{in}, buf_{wei}, and buf_{out} and the computing core work in parallel. In each iteration of the accelerator, a matrix multiplication is processed. The streaming performance for buf_{in}, buf_{wei}, and buf_{out} is determined by the bandwidth of the streaming interface. As the buffers are partitioned according to PE array's parallelism, there is no data fetch delay for the computing core. Benefiting from the pipeline design, the computing latency (Lat_{comp}) in an iteration is determined by three aspects, pipeline depth $Pipe_{depth}$ (Lat_{unit}), the number of executions for an output column (R_{WEI}/N_{PE1}), and the number of output columns (C_{WEI}). The computing latency for an accelerator iteration is described by Equation (4.20). At the run-time, the computing latency and streaming latency may vary from iteration to iteration as the matrices have different sizes.

$$
\begin{aligned}
Lat_{unit} &= Pipe_{depth} = \log_2 N_{PE1} + 2 \\
Lat_{comp} &= C_{WEI} * ceil(\frac{R_{WEI}}{N_{PE1}}) + Lat_{unit}
\end{aligned}
\tag{4.20}
$$

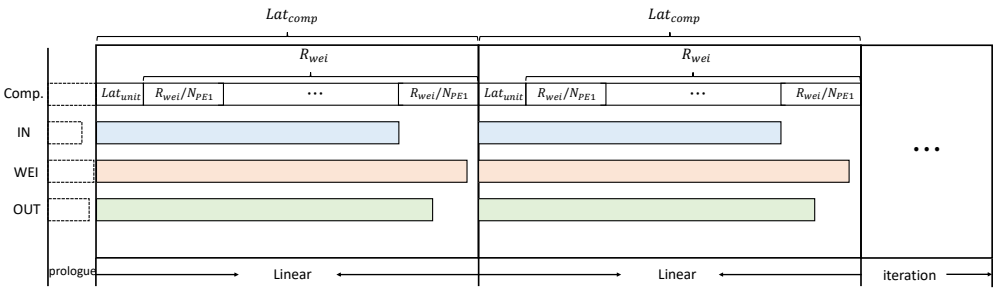


Figure 51: The accelerator running schedule.

4.2.4.3 Accelerator Analytical Model

As the parameters of attention mechanism vary in different applications while the DSP and LUT resources are flexible among FPGAs, an analytical model is developed to determine the design parameters of the accelerator and to analyze the system performance. The analytical model gives an estimation of the accelerator resource utilization and performance. The parameters in Figure 46 are used to illustrate the analytical model. The number of $PE1$ (N_{PE1}) is determined by DSP based $PE1$ (N^{dsp}_{PE1}) and LUT based $PE1$ (N^{lut}_{PE1}). In this model, the parallelism inside $PE1$ is set to the buf_{in}'s first dimension size N ($R_{IN} = N$). First, the N^{dsp}_{PE1} can be determined as the number of DSPs in an FPGA is fixed. After building the $PE1$ array in DSP, the number of $PE2$ can be determined ($PE2$ hierarchies=$\log_2 N^{dsp}_{PE1} + 1$). Next, if the FPGA LUT resource is still available, more LUT can be used in building $PE1$ and $PE2$ as INT8 operation is also efficient with FPGA LUT resources. The accelerator computing core's key parameters are summarized in Equation (4.21).

$$\begin{aligned} N_{PE1} &= N^{dsp}_{PE1} + N^{lut}_{PE1} \\ N^{dsp}_{PE1} &= floor(2 * N_{dsp}/N) \end{aligned} \tag{4.21}$$

The latency of the computing core is determined by the depth of the computing core which is $\log_2 N_{PE1} + 2$. The pipeline design and efficient data flow in the computing core achieve the minimum pipeline interval (II=1) to maximize the throughput. The run-time latency Lat^*_{comp} to process a matrix at the run-time is determined as $Lat^*_{comp} = C^*_{WEI} * ceil(\frac{R^*_{WEI}}{N_{PE1}}) + Lat_{unit}$, in which, C^* and R^* is the real size of the loaded weight at the run time.

As a result, the accelerator performance in processing a matrix can be modeled in Equation (4.22). The bottleneck is determined by the worst performance among the computing core and the buffer transmission. In Equation (4.22), Lat^{in}_{trans} can be determined by $Lat^{in}_{trans} = R^{*}_{IN} * C^{*}_{IN} * 8/B_{in}$, in which, B_{in} represents the streaming bandwidth assigned to buf_{in}. The Lat^{wei}_{trans} and Lat^{out}_{trans} can be acquired similarly. Then, the system performance Lat_{sys} is determined by Lat_{comp} and Lat_{trans}.

$$\begin{aligned} Lat_{trans} &= max(Lat^{in}_{trans}, Lat^{wei}_{trans}, Lat^{out}_{trans}) \\ Lat_{sys} &= max(Lat_{comp}, Lat_{trans}) \end{aligned} \tag{4.22}$$

While the proposed computing core achieves ultimate parallelism and pipeline performance, an adequate streaming bandwidth between off-chip and on-chip is also important. To avoid the system stall caused by data transmission, the minimum bandwidth requirement of each interface is also analyzed. After building the computing core and corresponding buffers, the required streaming bandwidth can be determined as $B^{min}_{in} \geq R_{IN} * C_{IN} * 8/Lat_{comp}$, $B^{min}_{wei} \geq R_{WEI} * C_{WEI} * 8/Lat_{comp}$, and $B^{min}_{out} \geq R_{OUT} * C_{OUT} * 8/Lat_{comp}$ for buf_{in}, buf_{wei}, and buf_{out}, respectively.

4.2.5 Experiment

4.2.5.1 Experiment Setup

We use two datasets from language translation, i.e., the Multi30K [91] (which is the subset of WMT2014) and the IWSLT'17 [92], to evaluation the accuracy impact brought by our proposed memory footprint aware compression. The key parameters of Transformer are listed in Table 10 which are consistent with the settings in [47, 82, 83]. The Transformer consists of 6 encoders and 6 decoders. The attention mechanism consists of 8 heads with d_k of 64 and d_{model} of 512. The compression and training process are performed by using Pytorch. The evaluation is performed in English-German (En-De) and German-English (De-En) translation for both datasets. The accelerator is built via Vivado HLS (v2019.1) and implemented via Vivado on SoC FPGA board ZCU102.

4.2.5.2 Model Compression Performance

The evaluation is performed in English-German (En-De) and German-English (De-En) translation for both datasets. The Transformer is trained to get the baseline accuracy (BLEU) first and then is applied with compression. The baseline and compression results such as compression ratio, BLEU score, and BLEU after quantization are shown in Table 11. In this table, one can observe that our method prunes 80% of the weights in Transformer for both En-De and De-En tasks in Multi30K without accuracy drop. When comparing with the state-of-the-art Transformer compression works including HAT [83] and Tran.Zip [82], we achieve near 10% and 30% higher pruning ratio, respectively. The pruned and trained model also adopts post-training quantization. The sparse and trained model is quantized to INT8 in Pytorch without accuracy loss. For dataset IWSLT'17, we achieve 70% and 75% compression for En-De and De-En tasks, respectively. In the studied machine translation tasks, the proposed memory footprint aware compression achieves 92.5-95% compression ratio in different datasets, reducing the Transformer weights to 8.8MB.

As a result of the memory footprint aware compression, the upper bound of the size of weights Q^w, K^w, V^w, O^w, $FFN1^w$, and $FFN2^w$ within Transformer is 50% of its original size. This is due to the coarse grained compression prunes half of the columns of each weight. Therefore, the buffer size for the six types of weight is controlled, and the shape is friendly to hardware buffer allocation.

Table 10: Transformer parameters

Model	N_{enc}	N_{dec}	$head$	d_k	d_{model}
Transformer	6	6	8	64	512

4.2.5.3 Accelerator Performance

An accelerator in INT8 towards Transformer compression $ratio_{coarse} = 50\%$ is built with Xilinx HLS and synthesized in Vivado (v2019.1).

Computing core implementation. On ZCU102 FPGA, we build the computing core with 74 $PE1$ (50 N_{PE1}^{dsp} and 24 N_{PE1}^{lut}) and 73 $PE2$. The parallelism of each PE is 100 which is the

Table 11: Performance of compression and quantization

			# Param (MB)	Compression Ratio (%)	BLEU	Quantized BLEU	Compression Ratio* (%)
Multi30K	En-De	Transformer	176	-	28.9	-	-
		HAT [83]	48	73% (-7%)	28.4	-	-
		TransformerZip [82]	86	50% (-30%)	26.4	-	-
		Ours	**8.8**	**80%**	**29**	**29**	**95%**
	De-En	Transformer	176	-	26	-	-
		Ours	**8.8**	**80%**	**25.8**	**25.8**	**95%**
IWSLT'17	En-De	Transformer	176	-	13	-	-
		Ours	**13.2**	**70%**	**13**	**13**	**92.5%**
	De-En	Transformer	176	-	15.5	-	-
		Ours	**11**	**75%**	**15.5**	**15.5**	**93.75%**

maximum sentence length of the dataset. In this design, the ideal "adder-tree" is fine-tuned in selected hierarchies to fit 74 $PE1$ outputs into $PE2$ array's data flow, resulting in a 10 stage pipeline with pipeline interval II=1. The resource utilization breakdown for $PE1$ and $PE2$ is shown in Table 12. 50 DSPs are used in building a $PE1^{dsp}$ and 5000 LUTs are used in building $PE1^{lut}$. 1500 LUTs are used in building a $PE2$. The developed computing core fully utilizes the FPGA DSP and LUT resources and can conduct 7400 multiply-accumulate operations per cycle. The theoretical throughput performance of the computing core is 2.22 Tops at 150 MHz.

Table 12: Processing element resource breakdown

	Number	**Parallelism**	**DSP**	**LUT**	**FF**	**BRAM**
$PE1^{dsp}$	50	100	50	-	1608	0
$PE1^{lut}$	24	100	-	5000	1608	0
$PE2$	73	100	-	1500	2400	0
Computing core	1	7400	2500	-	-	0

Accelerator buffer allocation. The buffer allocation is conducted by considering both the computing core and the compressed Transformer size. The parameters of the allocated buffer buf_{in}, buf_{wei}, and buf_{out} in the accelerator are shown in Table 13. The buffers are implemented by BRAMs and LUTRAMs. Since 74 $PE1$ are built, the first dimension of

buf_{wei} and the second dimension of buf_{in} is rounded to 518 for ease of PE array access. The second dimension of buf_{wei} is selected as 256 since the $ratio_{coarse}$ is 50%. After building the accelerator, the rest BRAMs on ZCU102 FPGA is utilized to expand the buf_{wei} in its second dimension. The expansion size of buf_{wei} is 2048, which can store 8.4Mb weights on the chip. At the interface between ZCU102 on-chip memory and the off-chip DDR, four streaming buses (128bit each) are available. Therefore, the streaming interface is determined according to buffer size: one streaming bus for buf_{in}; two streaming buses for buf_{wei}; one streaming bus for buf_{out}. The streaming bus allocation is represented by $\langle 1, 2, 1\rangle$ in the table.

Table 13: Accelerator buffer allocation

	$\boldsymbol{buf_{in}}$	$\boldsymbol{buf_{wei}}$	$\boldsymbol{buf_{out}}$
Rows	100	518	100
Columns	518	(256+2048)	256
Stream. Bus	$\langle 1, 2, 1\rangle$		

Table 14: Accelerator performance on ZCU102

Resource					Bus. $\langle\rangle$	Freq. (MHz)	Comp. Through.	End2End Through.	Lat_{comp} (ms)	Lat_{sys} (ms)
	DSP	LUT	BRAM	FF						
Avail	2520	274080	912	548160						
design1	99.3%	91.8%	28%	29.3%	$\langle 1, 2, 1\rangle$	150	1.87Tops	794Gops	0.014	0.033
design2	99.3%	91.9%	77%	26.4%	$\langle 1, 2, 1\rangle$	125	1.7Tops	845Gops	0.015	0.031
design2*	99.3%	91.9%	77%	26.4%	$\langle 2, 0, 2\rangle$	125	1.7Tops	1.4Tops	0.015	0.019

Table 15: Transformer accelerator performance model accuracy analysis.

Design	Our Model				On-Board				Deviation			
	Lat_{comp}	Lat_{sys}	BRAM	DSPs	Lat_{comp}	Lat_{sys}	BRAM	DSPs	Lat_{comp}	Lat_{sys}	BRAM	DSPs
Design1	1802	4096	-	2500	2100	4650	256	2503	14%	11.9%	-	0.1%
Design2	1802	3238	-	2500	1875	3875	702	2503	3.9%	16.4%	-	0.1%

Accelerator performance analysis. Three versions of the accelerator are implemented to show the superiority of our design. Design1 is built with no buf_{wei} expansion. Design2 is

built with buf_{wei} expansion. Design2* is built with buf_{wei} expansion and streaming interface tuning. In the three designs, we show the computing core's peak throughput (Computing Throughput), the accelerator system throughput considering data transfer (End2End Throughput), the computing latency (Lat_{comp}) in processing a unit matrix multiplication (buf_{in}[100][518], buf_{in}[518][256], and buf_{out}[100][256]), and the end-to-end latency (Lat_{sys}) in processing the unit matrix multiplication.

Design1 explores the highest throughput that the computing core can achieve. With the minimum active buffer, it achieves the highest working frequency at 150 MHz working in pure streaming in, computing, and streaming out mode (stream-in processing). It fully utilizes the DSP and LUT resource to build computing core, and 28% of BRAM resource to build buffers. The streaming interface is assigned as $\langle 1, 2, 1 \rangle$. At the run-time, design1's computing throughput is measured at 1.87 Tops. This is only slightly lower than its theoretical value due to the initialization of FPGA IP cores. The end-to-end processing throughput is 794 Gops. The latency Lat_{comp} is 0.014 ms and the Lat_{sys} is 0.033 ms. The significantly lowered end-to-end throughput is due to the limited streaming interface bandwidth, which causes significant computing core stall. If the bandwidth is sufficient to support the data consumption of the computing core, the design1 can work under 150 MHz with a throughput of 1.87 Tops.

Design2 explores the generic acceleration solution which partially stores the model on the chip. It supports "stream-in processing" and "in-situ processing" modes, in which, "in-situ processing" consumes the on-chip weights. Compared with design1, design2 utilizes the spare BRAM resource to expand the buf_{wei} capacity and 77% BRAMs are utilized. Therefore, 8.5Mb spare buffer spaces are generated to keep selected weights on the chip. With more active buffer, design2 works under 125 MHz. By partially storing a model on the chip, only the unbuffered weights need to be streamed in. The streaming interface is assigned as $\langle 1, 2, 1 \rangle$ to support both modes. We report "in-situ processing" performance in the table. The computing throughput is 1.7 Tops. The end-to-end throughput is 845 Gops. The latency Lat_{comp} is 0.015 ms. The Lat_{sys} is 0.031 ms. In this mode, the end-to-end system performance is moderately improved as buf_{wei} still occupies the streaming interface, leaving limited streaming bandwidth for other buffers. Therefore, design2's the overall performance can benefit from less transmission with the on-chip weights.

Design2* explores the accelerator solution for the small model that can be fully stored on the chip, which works in pure "in-situ processing". Design2* tunes the streaming interface to $\langle 2, 0, 2 \rangle$, more streaming bandwidth is assigned to buf_{in} and buf_{out} as no weight transfer is needed. After eliminating weight transfer, the end-to-end processing throughput is greatly improved to 1.4 Tops. The small gap between end-to-end throughput and computing throughput is due to the transmission initialization between DDR and FPGA.

The performance model accuracy analysis is evaluated in Table 15. As BRAM utilization can hardly been modeled for INT8 in FPGA, we show the performance model variation for latency and the DSP usage. We can see that the DSP usage can be successfully estimated and the performance model's latency variation is higher than the previous proposed Convolution and LSTM accelerator. This is due to the Transformer accelerator is assigned one matrix per execution, leading to less transmission workload. Thus, the uncertainty in the DDR would show more significant impact on the whole system.

Accelerator performance on real-life Transformer. We accelerate the compressed Transformer models discussed in Table 11, in which, 80%, 75%, and 70% compression ratios are achieved. This reduces the number of multiply–accumulate operations to 1.29, 1.65, and 2.04 Giga Operations (Gop). Due to the size of the compressed models, design2 is adopted to accelerate the three compressed models. The breakdown of accelerator resource utilization is listed in Table 16. In processing a full encoder layer to decoder layer inference, we achieve the latency of $6.8ms$, $7.2ms$, and $8.4ms$ for the three compression degrees. In this table, we also list the performance of the state-of-the-art Transformer accelerator [21] and [22]. [21] accelerated a shallow Transformer with 5.76MB weights and 0.205 Gop. Their accelerator is built on VCU118 FPGA in fix-point data via Vivado HLS, in which, 5647 DSPs are utilized. It achieves $2.94ms$ in accelerating the shallow Transformer. We compare the FPGA area efficiency via $Gop/Latency/N_{dsp}$ for fair comparison. As shown in the table, our accelerator achieves 6.3x, 7.6x, and 8.1x better efficiency at different compression ratios, which is significantly higher than the state-of-the-art accelerator [21]. Design [22] is built on XCVU13P FPGA in INT8 via hardware description language (HDL). The performance is measured by simulation in Vivado. We report its performance in accelerating a full Transformer, which achieves $8.9ms$ in latency. Its latency is significantly higher than our design. Furthermore,

the design [22] can hardly utilize its device resource, causing severe resource under-utilization (27.% LUT utilization, 36.5% BRAM (block-RAM) utilization, and 2% DSP utilization).

Table 16: End-to-end accelerator performance on real Transformer

Standard Transformer (Our Work)								
Sparsity	Gop	DSP	LUT	BRAM	FF	Freq. (MHz)	Latency	Area Efficiency
80%	1.29	2500	251725	699	142723	125	6.8 ms	0.076 (**6.3x**)
75%	1.65						7.2 ms	0.092 (**7.6x**)
70%	2.04						8.4 ms	0.097 (**8.1x**)
Shallow Transformer (Existing design[21])								
-	0.205	5647	268933	-	304012	-	2.94 ms	0.012
Standard Transformer (Existing design[22])								
-	-	129	471563	498	217859	200	8.9 ms	-

4.3 Summary

The proposed LSTM and Transformer accelerator with the associated performance models can efficiently address the large memory footprint size as well as the complex computation patterns of the model. The proposed analytical model can accurately and efficiently determine the accelerator parameters and the performance of the accelerator.

5.0 Multi-modal Multi-task Model to Multi-accelerator Mapping

This chapter presents the multi-modal multi-task model to multi-accelerator system mapping algorithm with computation and communication awareness. Besides the algorithm, aiming at system-level formulation, simulation, and optimization, we build a system-level simulator for fast mapping optimization and performance estimation.

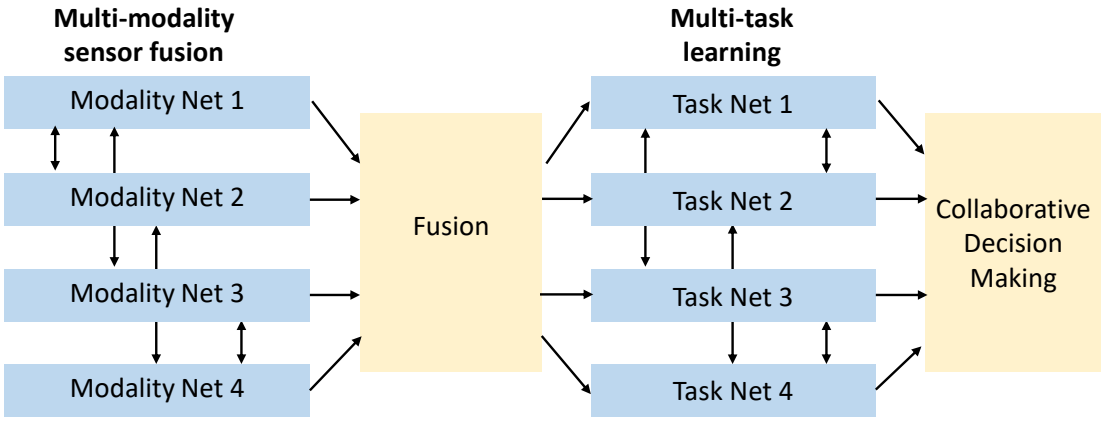


Figure 52: Multi-modal multi-task model.

5.1 Background

Multi-modal learning aims to process and relate information from multiple sources to capture the correspondences between modalities and gain an in-depth understanding of natural phenomena and usually focuses on modality representation, translation, alignment, fusion, and co-learning [1, 2]. In Virtual Reality (VR), the 'five senses': visual for sight, auditory for hearing, olfactory for smell, gustatory for taste, and haptic and thermal for touch, are necessarily be perceived to reconstruct, augment, and render the content [93]. This leads to the modalities such as images, videos, speech, text, etc. In autonomous system, sensors such as camera, radar, infrared, thermal, LiDAR, and gyro are placed to perceive the environment. The multi-modal multi-task structure naturally increases the computation heterogeneity and complexity, as well as the communication overhead. The difficulties of

the model can be summarized in three aspects: the multi-modal multi-task system-level heterogeneity; the layer heterogeneity in an individual modality/task net; the inter-block connection in the multi-modal multi-task caused model dependencies..

Model system-level heterogeneity. The multi-modal multi-task learning learns the features from different sources. Therefore, the architecture of different modality net is specialized for the target source. Figure 53 shows the existing multi-modal multi-task models for different applications. Figure 53 (a) shows a radar-image co-learning model, in which, the VGG16 network and Feature Pyramid Network are adopted as the backbones [31]. Figure 53 (b) shows a Lidar-image feature co-learning model, in which, Convolution, Sep Convolution, ASPP (Dilated Convolution), and LSTM layers are adopted to extract the features [30]. Figure 53 (c) shows a model for pose and odometry processing, in which, images are processed by Residual CNNs (ResNet [40] like network) [29]. Among different CNNs, the intermediate data in different tasks are shared between the Convolution layers. Figure 53 (d) shows an image-speed feature co-learning, in which, a Convolution and an FC-based backbone are adopted [28]. As shown in the existing models, the extreme model heterogeneity and inner-model dependency are the natural characteristics of an MMMT model.

Extreme heterogeneity in the modality and task net. A multi-modal multi-task model can be regarded as a superset of standalone DNN models. In general, even in one modality or task net, different sizes of Convolution, FC, Pooling, dilated Convolution, ReLu, Normalization, and Softmax is utilized to build a task specific model for a source data. The computing complexity and computing burden differ significantly even among the same type of layers as the layer parameters vary layer by layer. The diversity of DNN layers causes different computing patterns and accelerating efforts. The commonly adopted Convolution and FC are summarized in Figures 54a and 54b. The other modules such as Pooling and ReLu are less computation-intensive but own specific features.

Different layers have their own unique characteristics: Convolution is the most data computation and transmission intensive; FC is less computation-intensive but transmission-intensive; Pooling usually performs comparison in its function; Dilated Convolution is data computation and transmission intensive; ReLU only performs comparison in its function; Normalization and SoftMax is computation complex but less computation and transmis-

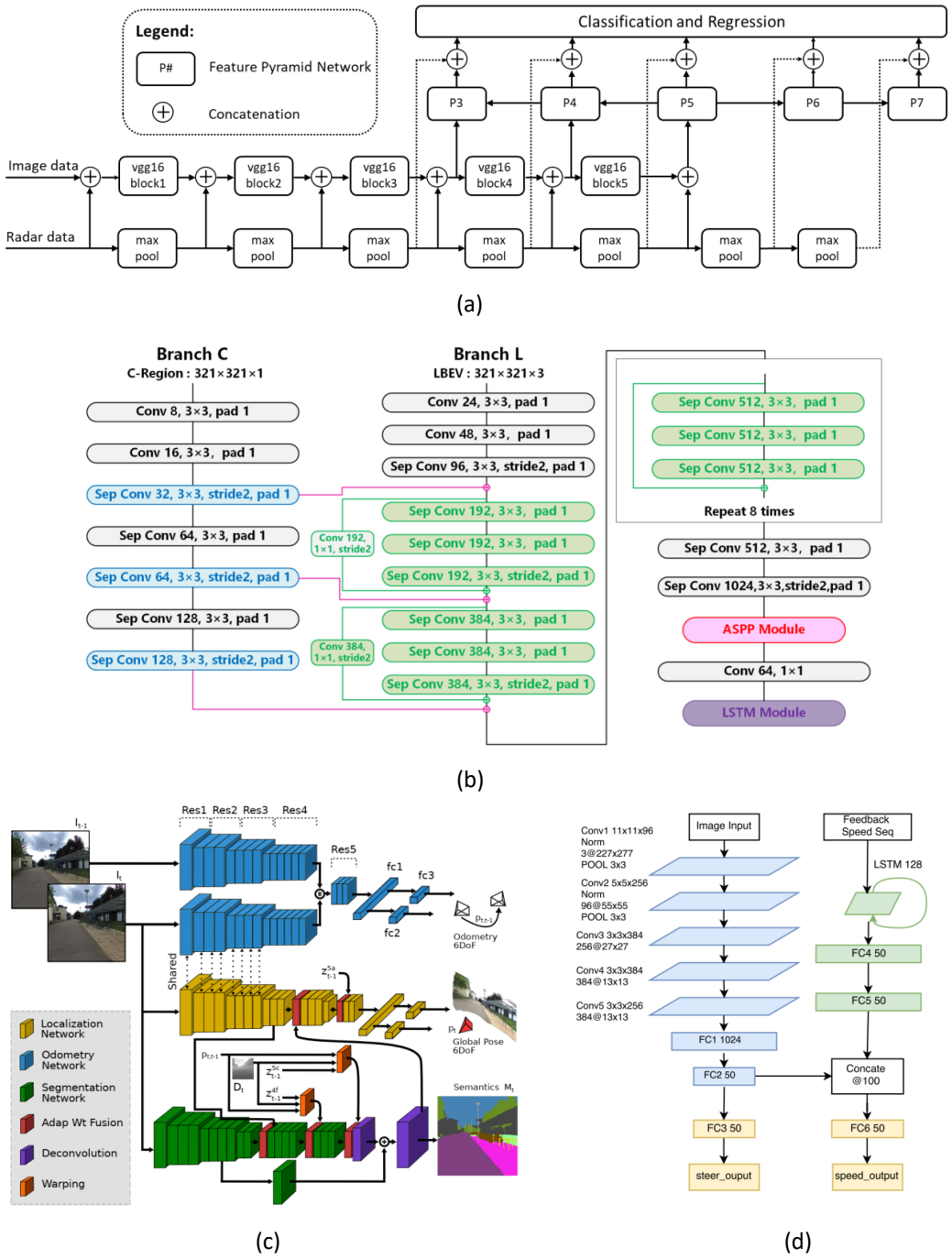
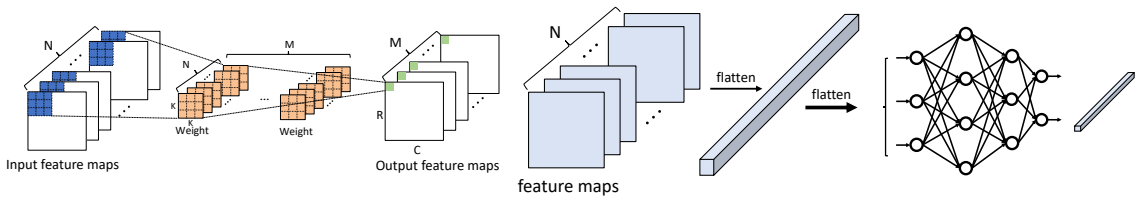


Figure 53: The existing multi-modal multi-task models.

sion intensive. As a result, accelerating layers such as Convolution, FC, Pooling, Dilated CNN, ReLu, Normalization, and SoftMax on their computing and memory pattern preferred accelerators can maximize the hardware computing efficiency. The existing DNN layer het-

erogeneity leads to accelerator efficiency differences such as utilization ratio, latency, and throughput. Furthermore, the inter-block sharing in the multi-modal multi-task model bring addition logic dependency between these modules and running order of the accelerators.



(a) Convolution layer (b) Fully-connected layer

Figure 54: Convolution and Fully-connection layer

Extreme heterogeneity in accelerators. The existing DNN accelerators are usually designed for specific DNN model layers. The change of the layer parameters can easily cause accelerator performance fluctuation as the computing pattern and transmission pattern in an accelerator are specialized. Therefore, the changes in characters such as Convolution IFM/OFM channels, weight filter size K may lead to performance differences on an accelerator. As the NVDLA style architecture shown in Figure 17, it primarily deep Convolution IFM/OFM channel ($< N, M >$) optimized architecture. The computing pattern in such architecture is optimized for the high parallelism computing in IFM and OFM channels. Such architecture is observed in designs such as [14, 62, 63, 94]. Shi-dianao style architecture optimizes the output feature map parallelism in a row and column direction, which may benefit the Convolution with large size feature map in row and column direction [17, 94]. Other accelerators such as Eyeriss [15] and Google TPU [95] adopt similar architecture in processing element (PE) array style to Shi-dianao. However, Eyeriss optimizes the PE's local buffer arrangement, which may benefit in Convolution layers with heavy weight usage; TPU adopts systolic array style for the PE array, which will benefit in higher throughput and lower latency for generic multiply-accumulation operation.

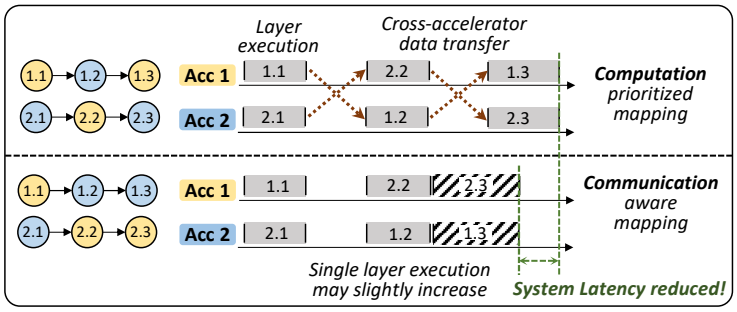


Figure 55: An example of communication-prioritized mapping and communication-aware mapping. The later slightly sacrifices the computation efficiency but reduces the overall system latency by avoiding expensive data movement.

Though the computing efficiency of normal DNN on the multi-accelerator system is addressed by [5, 6, 7, 8, 9, 10, 11, 23, 24], the emerging multi-modal multi-task DNNs model greatly increases the overall computation complexity as well as the communication overhead. A computation and communication aware mapping and scheduling of multi-modal multi-task DNNs to the multi-accelerator system is needed.

5.2 Motivation

Fig. 55 demonstrates the difference between computation-prioritized mapping and communication-aware mapping, where the former maps a layer purely based on its preferable computing and data flow pattern, while the latter slightly sacrifices computation efficiency but in turn, reduces the overall system latency by avoiding expensive data transfer. Modern multi-modal multi-task models tend to have more complex dependencies (e.g., the cross-layer connections in VlcoNet [35]), largely exaggerating the data transfer overhead.

Given the complexity of DNN models and heterogeneous systems, it is non-trivial to find a good H2H mapping that well balances **computation** and **communication**. First, the DNN accelerators are highly specialized for certain data flow. For instance, NVDLA [16] optimizes convolution channel-wise parallelism, while Shi-diannao [17] optimizes feature-map-wise parallelism. Model layers should be mapped to the accelerators with preferable

computation patterns to reduce computation latency. Most existing approaches highly prioritize the computation pattern matching but ignore communication [24]. Second, there are also communication-prioritized mapping algorithms [96] by forming task clusters and assigning a cluster to a processor. However, this will largely hurt the computing efficiency since the tasks within the same cluster do not necessarily run efficiently on the same accelerator. Meanwhile, the heavy cross-model dependency (*cross-talk*) in the heterogeneous models may also lead to ineffective clustering. Third, existing mapping algorithms lack the formulation of DNN models and system-level information such as accelerators' architecture and data flow. Without hardware awareness, such mapping algorithms are inefficient for H2H in the multi-accelerator system. Therefore, a hardware-aware mapping algorithm that considers both computation and communication simultaneously is needed.

The cornerstone for communication reduction is **data locality** by efficiently utilizing accelerators' local memory. Existing accelerators such as GPUs, ASICs, and FPGAs are attached with a local DRAM, which can be utilized to store model weights and to buffer intermediate activations of two adjacent layers to reduce cross-accelerator data movement. The challenge is that the computation-prioritized mapping can achieve the best efficiency per accelerator, but the overall performance may be compromised due to the transmission cost (and vice versa). Therefore, taking the multi-FPGA system as the vehicle, we propose a joint H2H optimization with both computation and communication awareness, considering the benefit of high data locality and suitable computation patterns simultaneously.

5.3 Methodology

5.3.1 System Formulation

Heterogeneous Models. A heterogeneous model has complicated dependencies especially for cross-talk connections. It is natural to formulate such a model as a direct graph $G_{model} = (V, E)$, where the vertices represent the layers and the edges represent the dependencies. In G_{model}, each node holds layer information such as Conv, FC, LSTM, etc., as well as their

Table 17: System performance modeling parameters

Acc Type	Parameters	Explanation
Conv	<N, M, R, C, K, S>	ofm_channel_num, ifm_channel_num, ofm_height, ofm_width, kernel_size, stride
FC	<N,M>	in_features, out_features
LSTM	<N,H,L>	in_size, hidden_szie, layers
–	W_{mem}	accelerator-to-host bandwidth
–	S_{mem}	private memory size

data dimension (e.g., feature map size). We consider three types of popular accelerators, Conv (Convolution), (FC) Fully-connection, and LSTM (Long short-term memory), with their layer parameters which are summarized in Table 17.

Heterogeneous System. We also formulate the multi-accelerator system as a directed graph $G_{sys} = \{G_{Acc_i}\}$, where each sub-graph G_{Acc_i} is a computation graph representing the layers' execution scheduling on the i-th accelerator Acc_i. Initially, each graph G_{Acc_i} is empty without any mapping. An example of G_{model} and initial G_{sys} with three initial empty Acc_i are shown in Fig. 56 input block. After mapping, the G_{acc} will be filled by nodes from G_{model} and the edges in G_{acc} indicate execution order.

In this work, we consider a multi-FPGA system proposed in [5], where each FPGA is connected to the host via network switches, enabling FPGA-to-FPGA and FPGA-to-host communication. The main memory at the host distributes the data to FPGAs' private memory (DRAM). The private memory capacity ranges from 512MB to 8 GB [97] and is usually used as additional buffers to mitigate the scarcity of FPGA on-chip memory. In general, the Ethernet speed ranges from 1 G to 10 G Ethernet (0.125 to 1.25 GB/s) in cloud-FPGA [6], and the FPGA private memory speed ranges from 6.4 GB/s to 460 GB/s [98]. The system-level configurable parameters include W_{mem} and S_{mem}, as shown in Table 17, referring to accelerator-to-host bandwidth and private memory size, respectively.

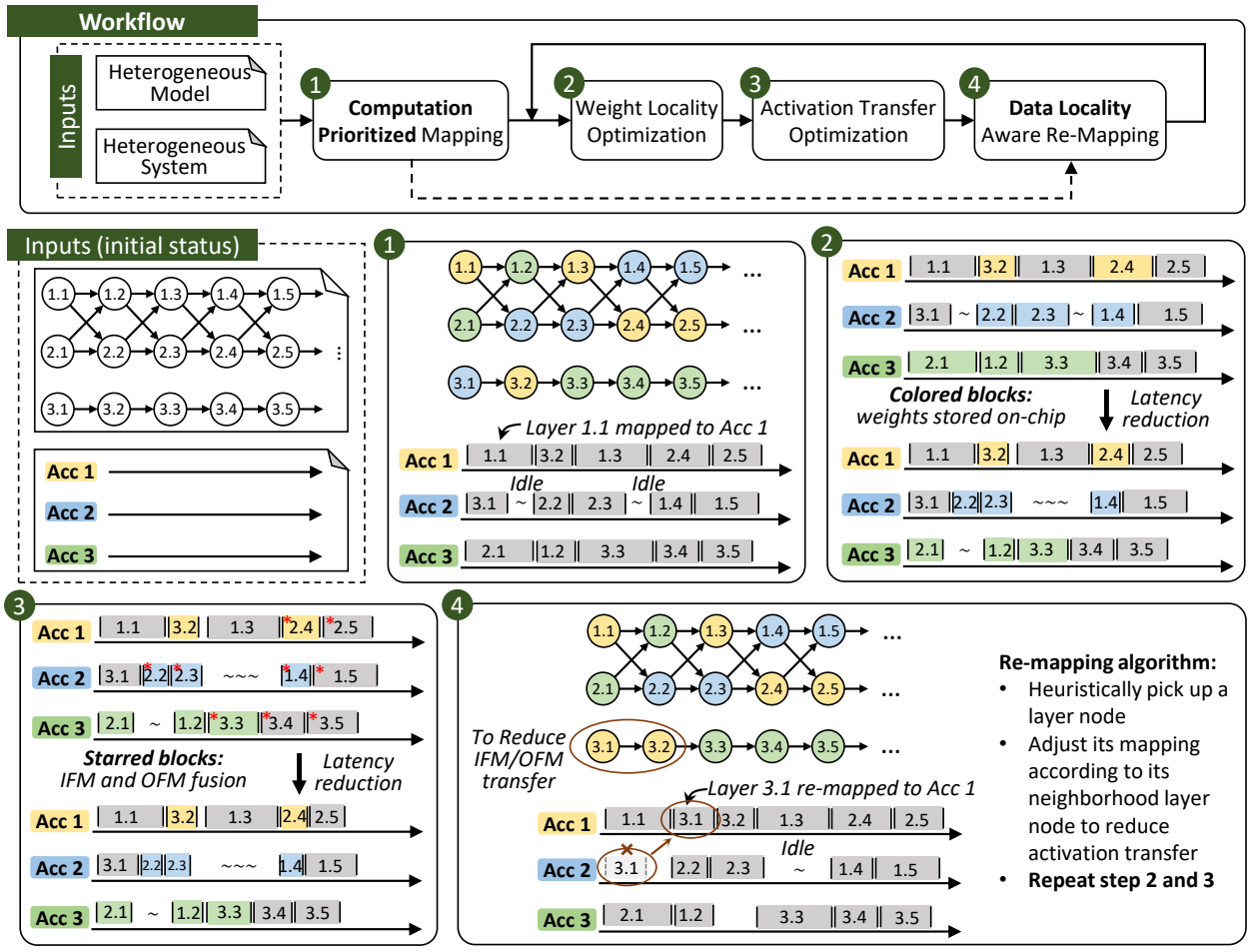


Figure 56: H2H mapping algorithm visualization. It includes 4 major steps: (1) computation-prioritized mapping; (2) weight locality optimization; (3) activation transfer optimization; (4) data locality aware remapping.

System Performance Model. We model the overall heterogeneous system performance at two levels: individual accelerators, and the overall system. First, for individual accelerators, there are plenty of analytical models P_{Acc} for different designs; we adopt the performance models from the prior works. For each accelerator, we consider the following configurable parameters: (1) W_{mem}, the accelerator to main memory bandwidth; (2) S_{mem}, the private memory size; (3) $Layer_{para}$, the layer parameters as shown in Table 17. For instance, the analytical model for the accelerator in [14] is $P_{Acc} =< W_{mem}, S_{mem}, Layer_{para} >$ with its with loop tiling setting $< R_{wei}, D_{type}, F_{acc}, W_{acc}, Tm, Tn, Tr, Tc >$. Second, for the system-level performance model, we develop simulator for the multi-FPGA system by allowing customizing the accelerator-to-host bandwidth as W_{mem}, which is also configurable by users.

5.3.2 Mapping Algorithm

The proposed mapping algorithm includes four steps. ① Computation-prioritized mapping. The heterogeneous ML model is mapped at layer-granularity, that each layer is mapped to the accelerator that best fits its computation data flow, ignoring all data movement optimizations (i.e., zero data locality). ② Weight locality optimization. Since each accelerator has its own private memory, we buffer part of the weights in the memory to maximally avoid weight data movement. ③ Activation transfer optimization. If two adjacent layers are mapped to the same accelerator, their intermediate activation, i.e., the output/input feature maps (OFM/IFM), will no longer need to transfer and thus latency can be reduced. ④ Data locality aware re-mapping. This step explores the trade-off between computation and communication, aiming to largely reduce communication cost with only slight computation efficiency degradation, which still results in overall performance improvement. The H2H algorithm flow is shown in Algorithm 4. It takes G_{model}, G_{sys}, and P_{Acc_i} as its inputs, and produce a mapped and scheduled solution with estimated system latency and energy.

Computation-prioritized Mapping. In the first step, we perform `Computation_Prioritized_Mapping`. It assigns the model layers to the accelerators that result in the best computation performance by assuming zero-private memory without any data locality at the accelerator. We use performance model P_{Acc_i} to estimate the latency of

a layer executing on the i-th accelerator, and assume that all the weights and intermediate results go to the main memory. To obtain the system overall latency, the layer scheduling on each accelerator must be determined. To guarantee a valid scheduling considering the layer dependencies especially across multiple sub-models, the algorithm determines the mapping and scheduling iteratively. In every iteration, it selects all the nodes without predecessors from G_{model} as a group, enumerates all possible mappings within the group (e.g., multiple nodes can be mapped to one or more accelerators), and selects the best one that results in the smallest system latency increment. .

An example is shown in Fig. 56 ①, where the color of the nodes represents which accelerator it is mapped to. The gray blocks represent the accelerator executions, where idle periods are introduced by layer dependency. Note that, in this step, we assume zero-private memory the latency values include both computation and communication: the latency of layer computation, weight transfer from the main memory, and IFM/OFM transfer from/to the main memory.

Weight Locality Optimization. `Weight_locality_Opt` is performed after computation prioritized mapping by utilizing the private memory for each accelerator. With private memory, weight transfer from main memory can be largely avoided, and it is a common practice to buffer part (or all) of the weights of the DNN layer(s) [97]. In this system, since multiple layers are mapped to the same accelerator, the layer weights must be selectively stored in the private memory, under a certain memory budget. Therefore, we propose to use the Knapsack algorithm to bind weights to accelerators to minimize the weight transfer. After weight binding, we first update each layer's latency, and then update the system's scheduling and overall latency. Note that, since the latency and scheduling change of one layer can affect its successor layers iteratively, we propose to update the layer scheduling *recursively*. This is especially efficient for graph structures by updating a node's direct successor neighbors without traversing the entire graph every time. The details of `Weight_Locality_Optimization` in building the simulator is shown in Algorithm 5. An example is shown in Fig. 56 2, where the colored blocks represent the layers whose weights are stored in the private memory with reduced latency. The system schedule is also updated with reduced overall latency.

Activation Transfer Optimization. After weight locality optimization, the activation (IFM/OFM) transfer will be optimized by `Activation_Transfer_Opt` to further reduce the overall cost. This is based on the assumption that, if two adjacent layers are mapped to the same accelerator, their intermediate IFM and OFM can be reused locally by taking advantage of the private memory and thus the activation transfer from/to the main memory can be avoided. We call it activation fusion. It is similar to weight locality optimization in a recursive manner: for each layer mapped to an accelerator, it checks its successor neighbors for activation fusion, updates its own and its neighbors' latency if applicable, and recursively updates the system's overall scheduling. The details of `Activation_Transfer_Opt` in building the simulator is shown in Algorithm 6. An example of activation fusion is shown in Fig. 56 3, where the starred blocks indicate the layers that are applicable for fusion.

Data Locality Aware Remapping. The weight and activation optimization are post-optimization for communication given a mapping solution. In this step, we strive for communication-oriented remapping (`Data_Locality_Remapping`), i.e., initial mapping tuning, aiming at largely reducing communication cost at the cost of slightly increased computation cost. Specifically, for each layer, we define a *remapping* operation that re-allocates a layer from its source accelerator to a new destination accelerator, on which its predecessors and/or successors are mapped to. This remapping reduces the activation transfer time but may increase the computation latency. The weight transfer latency may be increased or decreased depending on the available private memory capacity of the destination accelerator. Therefore, to determine the exact effect of a remapping operation, the weight locality and activation transfer optimization must be re-computed, i.e., the step 2 and 3 must be executed for every remapping attempt. We adopt the greedy algorithm and perform remapping attempt for every layer, and only accept such an attempt only if it reduces the system's overall latency, i.e., the benefit of communication reduction outweighs the computation cost increment. The algorithm terminates when there is no more layer that can be remapped with reduced overall latency. The details of `Data_Locality_Remapping` in building the simulator is shown in Algorithm 6.

An example of locality-aware remapping is shown in Fig. 56 ④. In this example, layer 3.1 is remapped from Acc2 to Acc1 since its neighbor layer 3.2 resides on Acc1, so that the

activation transfer between layer 3.1 and layer 3.2 can be reduced. Although in this example, the scheduling of layer 2.2 cannot move ahead because of layer dependency, in most cases, the source accelerator can reduce its latency because of the released memory budget for weights and can execute earlier because of a removed (remapped) layer.

5.3.3 Experiment

5.3.3.1 Experiment Setup

Heterogeneous models. Table 18 summarizes the 6 heterogeneous DNN models from different domains including AR/VR, Face Recognition, Sentiment Analysis, Activity Recognition, and Emotion Recognition. Most models use Convolution Neural Networks (ResNet, VGG, VD-CNN, and their variants) as the backbones, and there are typically 3 to 5 backbones placed together for MMMT learning with cross-backbone data dependencies.

Heterogeneous accelerators. We survey 12 state-of-the-art FPGA-based Convolution, FC, and LSTM accelerators and summarize them in Table 19. We replicate their performance models based on the original papers; we honor the private memory S_{mem} capacity based on the FPGA board used, ranging from 512 MB to 8 GB [97].

As the existing mapping algorithms are computation-prioritized that strive for finding the most suitable accelerators based on the data flow patterns [24]. Such mapping strategies are the same as the first step in our H2H mapping. To make a fair comparison, we take the results from H2H mapping after the second step including the weight locality optimization, since existing works can also assume private memory for the accelerators.

Table 18: Heterogeneous models

Domain	Model	Backbones	Para.
Augmented Reality	VLocNet [35]	ResNet-50 variants	192M
Face Recognition	CASUA-SURF [99]	ResNet-18 variants	13.2M
Sentiment Analysis	VFS [100]	VGG and VD-CNN variants	365M
Face Recognition	FaceBag [101]	ResNet variants	25M
Activity Recognition	CNN-LSTM [102]	ConvNet and LSTM variants	16M
Emotion Recognition	MoCap [36]	Convolution and LSTM unit	8M

5.3.3.2 Mapping performance

Latency and Energy Reduction. In Fig. 57 we present the latency and energy reduction from H2H mapping of the six heterogeneous DNN models. We test the mapping algorithm under different network bandwidth configurations (W_{mem}): Low- (0.125GB/s); Low (0.15GB/s); Mid- (0.25GB/s); Mid (0.5GB/s); High (1.25GB/s). The x-axis refers to the

Table 19: State-of-the-art FPGA DNN accelerators

Name	Accelerator Type	Optimization	FPGA
J.Z [103]	Convolution	On-chip memory	GX1150
C.Z [14]	Convolution	Channel parallel.	VC707
W.J [62]	Convolution	Memory and Channel	ZCU102
J.Q [104]	Conv/FC/(LSTM)	Computing Generality	ZC706
A.C [58]	Convolution	Loop Optimization	XC7Z045
Y.G [54]	Conv/FC/LSTM	Computing Generality	Stratix-V
T. M [12]	Convolution	Loop Optimization	GX1150
A.P [105]	Convolution	Winograd	Stratix-V
X.W [106]	Convolution	Systolic Array	GT1150
S.H [107]	LSTM/FC	Deep Pipeline	XCKU060
X.Z [65]	LSTM	Gate Parallelism	PYNQ-Z1/VC707
B.L [21]	LSTM	Deep Pipeline	VCU118

four steps in H2H mapping algorithm, and the y-axis is the "simulated" system latency in seconds and energy in joule. The H2H mapping algorithm achieves 15% to 74% system latency reduction and 23% to 64% energy reduction compared with the baseline mapping [24] when the system is bandwidth bounded, i.e., under the Bandwidth-Low- setting. With high bandwidth, the H2H still reduces overall latency by 10% to 50%. In half of the evaluated cases, we achieved over 60% latency reduction.

The detailed latency reduction after each step is shown in Table 20. Since we regard the second step as the baseline, we present the absolute latency values (in seconds) for steps 1 and 2 and the relative values for steps 3 and 4 compared with step 2. Apparently, when the bandwidth increases, the reduction decreases, but even with high bandwidth, network CNN-LSTM and MoCap still reduce latency by almost half from H2H mapping.

H2H performance analysis. In Fig. 58 (a), we visualize the communication and computation latency ratio using the mapping results under Bandwidth-Low- of the six models. Note that after H2H mapping, the computation ratio greatly increases (yellow bars), where MoCap increases from 21% to 94%, demonstrating that the communication overhead is largely reduced. We also show the H2H mapping algorithm execution time in Fig. 58. Per DNN model, the search time is consistently low across different models, less than one second. The VLocNet needs longer search time than others since it consists of 141 layers; the CNN-LSTM and MoCap need significantly less search time as they consist of less than 30 layers.

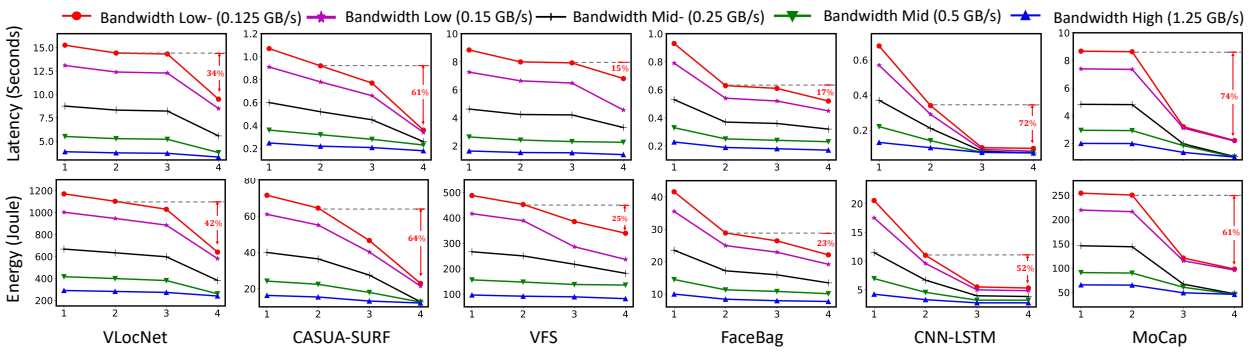


Figure 57: The latency and energy performance comparison.

5.3.3.3 Performance comparison and mapping uncertainty discussion

Performance comparison. The proposed computation and communication aware mapping can efficiently find an optimized mapping solution between the two heterogeneity. By heuristically performing mapping and re-mapping in the four steps, we found the optimized solution in seconds. To show the gap between the proposed method and the optimal solution, we enumerate all possible mappings and then schedule the execution. To enumerate all solutions, N^M mappings are generated and scheduled, in which, N represent the number of available accelerators in the heterogeneous system and M represents the number of layers in the heterogeneous model. As the surveyed heterogeneous model consists of up to 150 layers and the number of surveyed accelerators is 12, we perform the comparison by mapping the first 9 or 10 layers in different models, to a 2, 3, 4, and 5 accelerator system, under the system bandwidth setting of Low-. Table 22 and Table 23 show the performance comparison with the optimal solution acquired via enumerating all possible mappings in all six models. In this figure, the 'Optimal' represents the optimal solution and 'H2H' represent the proposed method. We can see that in the majority of the mapping situations, the H2H can achieve near-optimal performance. In the worst scenario, the H2H falls 2x˜3x behind the optimal solution. However, enumerating all possible solutions demands a great mass of time. For the 10-layer VLocNet to a 5-accelerator system, it takes 5.37 hours to find the optimal solution via enumeration while the H2H only takes 0.045 seconds. The proposed H2H achieves 4.3 million speed-up in searching time for VLocNet mapping. In all evaluated

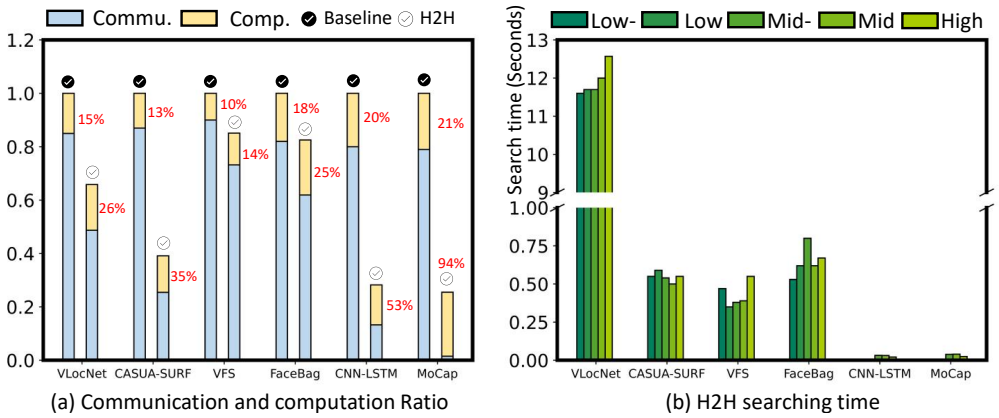


Figure 58: Communication and computation ratio.

cases, the proposed H2H can find the optimized solution in less than 0.1 seconds.

Uncertainty discussion. The robustness and efficiency of the H2H method are based on the established performance model of the accelerators in the system. During the mapping of H2H, we honor the accuracy of the surveyed accelerators and their performance models. However, our previous works show that the error between the performance model estimation and the real hardware performance can not be avoided. In our works, a maximum error rate of 6% is observed, in which, the performance model predicted latency is usually better than the hardware's due to the extra time cost at the run time in the accelerator's DDR. In this discussion, we regard such errors as the uncertainty of the performance. Such uncertainty is not considered during the H2H mapping. Therefore, we would analyze the impact from the uncertainty of the performance model after the H2H mapping. To evaluate the impact, under the system bandwidth setting of Low-, we take the mapping results of the H2H on the six heterogeneous models and re-schedule of system execution after applying a 6% performance uncertainty to each layer. Table 24 shows the system latency before and after applying the error, in which, H2H represents the H2H original performance and H2H* represents the performance with the 6% uncertainty. We can observe that in most cases, the system latency is linearly increased by 6% as the latency of each layer increases by 6%. In VLocNet, the system latency is increased by 10%, this is caused by the layer dependency in the heterogeneous model. The factor that causes higher system latency is visualized in Figure 59 In this case, layer 1.1 and layer 2.3 are layer 2.4's predecessors. Without apply-

Table 20: Latency reduction breakdown comparing with the second step (baseline).

Bandwidth	VLocNet				CASUA-SURF				VFS			
	1	2	3	4	1	2	3	4	1	2	3	4
Low-	15.27	14.43	99.31%	65.84%	1.07	0.92	83.70%	39.13%	8.84	7.99	99.12%	85.11%
Low	13.10	12.40	99.11%	68.55%	0.91	0.78	84.62%	43.59%	7.26	6.64	97.59%	68.67%
Mid-	8.76	8.33	98.80%	66.87%	0.60	0.52	86.54%	50.00%	4.63	4.24	99.29%	78.07%
Mid	5.51	5.28	98.67%	71.78%	0.36	0.32	87.50%	71.88%	2.64	2.42	95.45%	93.39%
High	3.88	3.76	98.67%	88.03%	0.25	0.22	94.55%	81.82%	1.64	1.53	98.69%	90.20%

Table 21: Latency reduction breakdown comparing with the second step (baseline).

Bandwidth	FaceBag				CNN-LSTM				MoCap			
	1	2	3	4	1	2	3	4	1	2	3	4
Low-	0.93	0.63	96.83%	82.54%	0.68	0.34	29.41%	28.24%	8.67	8.63	37.08%	25.49%
Low	0.79	0.54	96.30%	83.33%	0.57	0.29	31.03%	27.59%	7.39	7.35	42.18%	29.39%
Mid-	0.53	0.37	97.30%	86.49%	0.37	0.21	38.10%	33.33%	4.82	4.8	40.83%	21.67%
Mid	0.33	0.25	96.00%	92.00%	0.22	0.14	52.14%	52.14%	2.94	2.92	62.67%	34.25%
High	0.29	0.189	95.24%	89.95%	0.13	0.10	73.00%	70.00%	1.99	1.98	67.68%	50.51%

ing the uncertainty, the system critical path lies on the blue accelerator starting from layer 2.1 to 2.5. After applying the uncertainty, due to the longer execution time of 1.1 and the dependency between 1.1 and 2.4, a running stall of layer 2.4 on the blue accelerator would exist. In the figure, the red block represents the additional 6% latency after applying the uncertainty. As a result, the critical path in the blue accelerator would be influenced by the uncertainty of layer 1.1, causing a non-linear system latency increase.

Table 22: The mapping performance and searching time performance comparison.

		VLocNet*		CASUA-SURF*		VFS*	
# of Acc.	Method	Latency(s)	Search time	Latency(s)	Search time	Latency(s)	Search time
2	Optimal	0.67	1.53s(0.017x)	0.12	0.63s(32x)	0.43	1.49s(114x)
	H2H	1.13	0.017s	0.12	0.02s	0.76	0.013s
3	Optimal	0.549	87.72s(6.3e3x)	0.076	24.79s(1.9e3x)	0.355	86.73s(4.3e3x)
	H2H	0.56	0.014s	0.076	0.013s	0.42	0.02
4	Optimal	0.549	0.56h(6.7e4x)	0.076	433s(2.1e4x)	0.257	0.57h(5.2e4x)
	H2H	1.06	0.03s	0.12	0.02s	0.53	0.039s
5	Optimal	0.549	5.37h(4.3e5x)	0.071	0.91h(9.4e4x)	0.22	5.2h(3.3e5x)
	H2H	1.03	0.045s	0.21	0.035s	0.25	0.057s

Table 23: The mapping performance and searching time performance comparison.

		FaceBag*		CNN-LSTM*		MoCap*	
# of Acc.	Method	Latency(s)	Search time	Latency(s)	Search time	Latency(s)	Search time
2	Optimal	0.011	1.52s(89x)	0.058	1.56s(71x)	0.205	0.74s(27x)
	H2H	0.015	0.017s	0.065	0.022s	0.73	0.027s
3	Optimal	0.006	88.87s(386x)	0.058	90.45s(3e3x)	0.205	28.72s(1.2e3x)
	H2H	0.012	0.023s	0.14	0.028s	0.63	0.024s
4	Optimal	0.006	0.56h(5.7e4x)	0.058	0.58h(6.7e4x)	0.205	494s(1.7e4x)
	H2H	0.011	0.035s	0.073	0.031	0.205	0.028s
5	Optimal	0.0068	5.3h(4.4e5x)	0.058	5.43h(4.4e5x)	0.205	1h(7.2e5x)
	H2H	0.015	0.043s	0.1	0.044s	0.205	0.005s

Table 24: The mapping uncertainty analysis.

Model	**VLocNet**			**CASUA-SURF**			**VFS**		
Method	H2H	H2H*	Sys error	H2H	H2H*	Sys error	H2H	H2H*	Sys error
latency	9.5s	10.5s	10%	0.36s	0.44s	6%	6.8s	7.23s	6%
Model	**FaceBag**			**CNN-LSTM**			**MoCap**		
Method	H2H	H2H*	Sys error	H2H	H2H*	Sys error	H2H	H2H*	Sys error
latency	0.52s	0.55s	6%	0.096s	0.114s	6%	2.2s	2.33s	6%

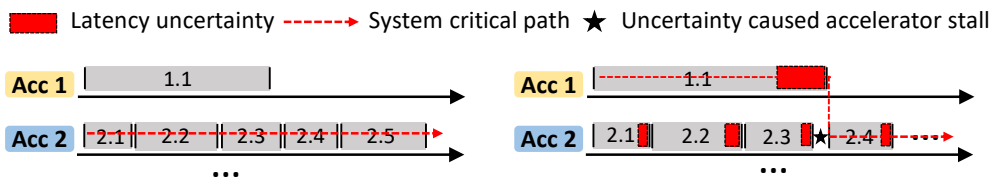


Figure 59: The visualization of system latency non-linear increment

Algorithm 4: H2H Mapping and Scheduling

Input: G_{model}, $G_{sys} = \{G_{Acc_i}\}$, P_{Acc_i}

Output: G^*_{model}, $G^*_{sys} = \{G^*_{Acc_i}\}$, $Sys_{latency}$, Sys_{energy}

Function `Computation_Prioritized_Mapping()`:
  for *nodes in* G_{model} *without predecessors* **do**
    Enumerate all possible mappings based on P_{Acc_i}
    Choose the mapping with minimum $\Delta Sys_{latency}$
Function `Weight_Locality_Opt()`:
  `Knapsack_Solver`(G_{model}, G_{sys})
  $Sys_{latency}, Sys_{energy} \longleftarrow$ `update_System_Scheduling()`
Function `Activation_Transfer_Opt()`:
  for *every node pair adjacent in* G_{sys} **do**
    `activation_Fusion`(*node pair*)
  $Sys_{latency}, Sys_{energy} \longleftarrow$ `update_System_Scheduling()`
Function `Data_Locality_Remapping()`:
  Repeat
    for *every* $n \in G_{model}$ **do**
      Attempt to remap n to its neighbors' Acc
      `Weight_Locality_Opt()`
      `Activation_Transfer_Opt()`
      $\Delta Sys_{latency} \leftarrow$ `update_System_Scheduling()`
      Accept remap if $\Delta Sys_{latency} < 0$
  Until *no more beneficial remapping operations*;

Algorithm 5: Weight Locality Optimization

Input: G_{model}, G_{sys}, P_{Acc_i}.

Output: G^*_{model}, G^*_{sys}, $Sys_{latency}$, Sys_{energy}.

G^{bind}_{sys} = KnapsackSolver(G_{sys}, P_{Acc_i})

$G^{tmp}_{sys} = G^{bind}_{sys}$;

while $node_{bind}$ ***exists in*** G^{tmp}_{sys} **do**

- node = findMinNode(G^{tmp}_{sys})
- G_{model}.updateNode(node)
- G_{sys}.updateNode(node)
- succesors = Successors(G_{model}, node) + Successors(G_{sys}, node)
- **for** *succ in successors:* **do**
  - `NodeShorten`(succ)
- G^{tmp}_{sys}.remove(node)

Function `NodeShorten`(*node*)**:**

- preds = Predecessors(G_{model}, node) + Predecessors(G_{sys}, node)
- predEndTime = Max(preds.endTime)
- **if** *predEndTime* $<$ *node.startTime* **then**
  - $G_{model}.update(node, predEndTime)$
  - $G_{sys}.update(node, predEndTime)$
  - succesors = Successors(G_{model}, node) + Successors(G_{sys}, node)
  - **for** *succ* ***in*** *succesors* **do**
    - `NodeShorten`(succ)
- **return**

Algorithm 6: Activation Transfer Optimization

Input: G_{model}, G_{sys}, P_{Acc_i}.

Output: G^*_{model}, G^*_{sys}, $Sys_{latency}$, Sys_{energy}.

accInit= G_{sys}.nodesNoPredecessor()

for *node* ***in*** *accInit* **do**
    `checkFusion`(node)

$G^{tmp}_{sys} = G_{sys}$;

while G^{tmp}_{sys} ***Not*** *empty* **do**
    node = findMinNode(G^{tmp}_{sys})
    `NodesShorten`(node)
    G^{tmp}_{sys}.remove(node)

Function `checkFusion`(*node*)**:**
    pred = Predecessor(G_{sys}, node)
    succ = Successor(G_{sys}, node)
    try:
        checkDependency(pred, node, succ)
        $G_{sys}.update(node)$
        $G_{model}.update(node)$
    accept *if* $Sys^*_{latency} < Sys_{latency}$**:**
        G_{sys}, G_{model}
    `checkFusion`(succ)
    return

Algorithm 7: Data Locality Remapping

Input: G_{model}, G_{sys}, P_{Acc_i}.

Output: G^*_{model}, G^*_{sys}, $Sys_{latency}$, Sys_{energy}.

for *node* ***in*** G_{model} **do**
    `AccFineTune`(node)

Function `AccFineTune`(*node*)**:**
    neighbors = Predecessors(G_{model}, node) + Successors(G_{model}, node)
    for *neighbor* ***in*** *neighbors* **do**
        if $G_{sys}.node.acc \neq G_{sys}.neighbor.acc$ **then**
            try:
                `AccAssign`(node, neighbor)
            accept *if* $Sys^*_{latency} < Sys_{latency}$**:**
                G_{model}, G_{sys}
    for *succ* ***in*** *Successors(G_{model}, node)* **do**
        `AccFineTune`(succ)

Function `AccAssign`(*node, neighbor*)**:**
    $G_{model}.update(node, neighbor)$
    $G_{sys}.update(node, neighbor)$
    Call Algorithm (`Weight_Locality_Optimization`)
    Call Algorithm (`Activation_Transfer_Optimization`)
    if $Sys^*_{latency} < Sys_{latency}$ **then**
        accept G_{sys}, G_{model}
    return

6.0 Conclusion and future work

In this dissertation, we present a computation and communication aware mapping algorithm to enable an efficient heterogeneous ML model to heterogeneous multi-accelerator system mapping. With more and more real-life applications are adopting ML algorithms, the ML algorithm is developing from a single task level to an intelligent agent that can perceive the environment and make decisions, which makes heterogeneous models (multi-modal multi-task ML algorithm) trend in algorithm design. DNN models from domains such as Computer Vision, Natural Language Processing, etc., are integrated into a hybrid model. With the growth of the ML market, both academics and industry are exploring suitable accelerator architectures for different DNN models/layers. While researchers have been ambitiously building individual accelerators for the emerging DNN models, joint processing of the multi-modal multi-task ML algorithm with both computation and communication awareness is not explored.

To address the computation and communication efficiency of multi-modal multi-task ML algorithm, this thesis three techniques. First, the Convolution FPGA accelerator with its corresponding workload partition and running schedule is proposed to increase the computing efficiency is proposed. Besides, an accurate performance model is developed to formulate the accelerator architecture, estimate and quantify the accelerator performance and resource utilization. Second, the LSTM and Transformer FPGA accelerators are proposed to address the complex data flow and diverse computing patterns in the algorithm. The algorithm-hardware co-design is utilized to address the large size algorithm memory footprint and the computations. Third, based on the in-depth understanding of the FPGA DNN accelerator design, a computation and communication aware multi-modal multi-task model to multi-accelerator system mapping is proposed.

The proposed mapping method can efficiently find an optimized mapping solution between the heterogeneous model and heterogeneous system. We do observe that the performance gap exists between the proposed solution and the optimal solution in some cases. In the future, more efficient re-mapping algorithm will be explored to enhance the performance of the proposed method.

Bibliography

[1] Cong Hao et al. Software/hardware co-design for multi-modal multi-task learning in autonomous systems. In *Proceeding of AICAS*, pages 1–5. IEEE, 2021.

[2] Tadas Baltrušaitis, Chaitanya Ahuja, and Louis-Philippe Morency. Multimodal machine learning: A survey and taxonomy. *IEEE transactions on pattern analysis and machine intelligence*, 41(2):423–443, 2018.

[3] Murium Iqbal et al. A multimodal recommender system for large-scale assortment generation in e-commerce. *arXiv preprint arXiv:1806.11226*, 2018.

[4] Alexander Mehler et al. Vannotator: A framework for generating multimodal hypertexts. In *Proceedings of the Hypertext and Soc. Media*, pages 150–154. 2018.

[5] Jeremy Fowers et al. A configurable cloud-scale dnn processor for real-time ai. In *Proceeding of ISCA*, pages 1–14. IEEE, 2018.

[6] Aws network. `https://aws.amazon.com/blogs/aws/new-gigabit-connectivity-options-for-amazon-direct-connect/`.

[7] Hasitha Muthumala Waidyasooriya and Masanori Hariyama. Multi-fpga accelerator architecture for stencil computation exploiting spacial and temporal scalability. *IEEE Access*, 7:53188–53201, 2019.

[8] Nicola Cadenelli, Zoran Jaksić, Jordà Polo, and David Carrera. Considerations in using opencl on gpus and fpgas for throughput-oriented genomics workloads. *Future Generation Computer Systems*, 94:148–159, 2019.

[9] Saman Biookaghazadeh, Pravin Kumar Ravi, and Ming Zhao. Toward multi-fpga acceleration of the neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 17(2):1–23, 2021.

[10] Xu Liu, Hibat Allah Ounifi, Abdelouahed Gherbi, Yves Lemieux, and Wubin Li. A hybrid gpu-fpga-based computing platform for machine learning. *Procedia Computer Science*, 141:104–111, 2018.

[11] María Angélica Dávila Guzmán, Raúl Nozal, Rubén Gran Tejero, Maria Villarroya-Gaudo, Darío Suárez Gracia, and Jose Luis Bosque. Cooperative cpu, gpu, and fpga heterogeneous execution with enginecl. *The Journal of Supercomputing*, 75(3):1732–1746, 2019.

[12] Yufei Ma et al. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 FPGA*, 2017.

[13] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing cnn accelerator efficiency through resource partitioning. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 535–547. IEEE, 2017.

[14] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason 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*, pages 161–170, 2015.

[15] Yu-Hsin Chen et al. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*.

[16] Nvidia. Website. `http://nvdla.org/`.

[17] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 92–104, 2015.

[18] Andre Xian Ming Chang and Eugenio Culurciello. Hardware accelerators for recurrent neural networks on fpga. In *2017 IEEE International symposium on circuits and systems (ISCAS)*, pages 1–4. IEEE, 2017.

[19] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. Fpga-based accelerator for long short-term memory recurrent neural networks. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 629–634. IEEE, 2017.

[20] Sicheng Li, Chunpeng Wu, Hai Li, Boxun Li, Yu Wang, and Qinru Qiu. Fpga acceleration of recurrent neural network based language model. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 111–118. IEEE, 2015.

[21] Bingbing Li et al. Ftrans: energy-efficient acceleration of transformers using fpga. In *Proceedings of ISLPED*, pages 175–180, 2020.

[22] Siyuan Lu, Meiqi Wang, Shuang Liang, Jun Lin, and Zhongfeng Wang. Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer. *arXiv preprint arXiv:2009.08605*, 2020.

[23] Yao Chen et al. Cloud-dnn: An open framework for mapping dnn models to cloud fpgas. In *Proceedings of the 2019 FPGA*, pages 73–82, 2019.

[24] Hyoukjun Kwon et al. Heterogeneous dataflow accelerators for multi-dnn workloads. In *Proceedings of HPCA*. IEEE, 2021.

[25] Md Shad Akhtar, Dushyant Singh Chauhan, and Asif Ekbal. A deep multi-task contextual attention framework for multi-modal affect analysis. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 14(3):1–27, 2020.

[26] Yuenan Hou, Zheng Ma, Chunxiao Liu, and Chen Change Loy. Learning to steer by mimicking features from heterogeneous auxiliary networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 8433–8440, 2019.

[27] Jose Solomon and Francois Charette. Hierarchical multi-task deep neural network architecture for end-to-end driving. *arXiv preprint arXiv:1902.03466*, 2019.

[28] Zhengyuan Yang, Yixuan Zhang, Jerry Yu, Junjie Cai, and Jiebo Luo. End-to-end multi-modal multi-task vehicle control for self-driving cars with visual perceptions. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 2289–2294. IEEE, 2018.

[29] Noha Radwan, Abhinav Valada, and Wolfram Burgard. Vlocnet++: Deep multitask learning for semantic visual localization and odometry. *IEEE Robotics and Automation Letters*, 3(4):4407–4414, 2018.

[30] Ruochen Yin, Yong Cheng, Huapeng Wu, Yuntao Song, Biao Yu, and Runxin Niu. Fusionlane: Multi-sensor fusion for lane marking semantic segmentation using deep neural networks. *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[31] Felix Nobis, Maximilian Geisslinger, Markus Weber, Johannes Betz, and Markus Lienkamp. A deep learning-based radar and camera sensor fusion architecture for ob-

ject detection. In *2019 Sensor Data Fusion: Trends, Solutions, Applications (SDF)*, pages 1–7. IEEE, 2019.

[32] Nhu-Van Nguyen, Christophe Rigaud, and Jean-Christophe Burie. Comic mtl: optimized multi-task learning for comic book image analysis. *International Journal on Document Analysis and Recognition (IJDAR)*, 22(3):265–284, 2019.

[33] Narada Warakagoda, Johann Dirdal, and Erlend Faxvaag. Fusion of lidar and camera images in end-to-end deep learning for steering an off-road unmanned ground vehicle. In *2019 22th International Conference on Information Fusion (FUSION)*, pages 1–8. IEEE, 2019.

[34] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. *Introduction to linear regression analysis*. John Wiley & Sons, 2021.

[35] Abhinav Valada, Noha Radwan, and Wolfram Burgard. Deep auxiliary learning for visual localization and odometry. In *2018 ICRA*, pages 6939–6946. IEEE, 2018.

[36] Samarth Tripathi et al. Multi-modal emotion recognition on iemocap with neural networks. *arXiv preprint arXiv:1804.05788*, 2018.

[37] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[38] Selim Seferbekov, Vladimir Iglovikov, Alexander Buslaev, and Alexey Shvets. Feature pyramid network for multi-class land segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 272–275, 2018.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[41] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[42] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, October 2014. Association for Computational Linguistics.

[43] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[44] Matt W Gardner and SR Dorling. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment*, 32(14-15):2627–2636, 1998.

[45] Sankar K Pal and Sushmita Mitra. Multilayer perceptron, fuzzy sets, classifiaction. 1992.

[46] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[48] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx adaptive compute acceleration platform: Versaltm architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 84–93, 2019.

[49] Michael Ditty et al. Nvidia's xavier soc. In *Hot chips: a symposium on high performance chips*, 2018.

[50] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, et al. Compute solution for tesla's full self-driving computer. *IEEE Micro*, 40(2):25–35, 2020.

[51] Declan O'Loughlin, Aedan Coffey, Frank Callaly, Darren Lyons, and Fearghal Morgan. Xilinx vivado high level synthesis: Case studies. 2014.

[52] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. From opencl to high-performance hardware on fpgas. In *22nd international conference on field programmable logic and applications (FPL)*, pages 531–534. IEEE, 2012.

[53] Olaf Ronneberger et al. U-net: Convolutional networks for biomedical image segmentation. In *Proceeding of MICCAI*, pages 234–241. Springer, 2015.

[54] Yijin Guan et al. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *2017 IEEE FCCM*.

[55] Chen Zhang et al. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on TCAD*, 38(11):2072–2085, 2018.

[56] Mohammad Motamedi, Philipp Gysel, and Soheil Ghiasi. Placid: A platform for fpga-based accelerator creation for dcnns. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 13(4):1–21, 2017.

[57] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[58] Andre Xian Ming Chang et al. Compiling deep learning models for custom hardware accelerators. *arXiv preprint arXiv:1708.00117*, 2017.

[59] Atul Rahman, Sangyun Oh, Jongeun Lee, and Kiyoung Choi. Design space exploration of fpga accelerators for convolutional neural networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1147–1152. IEEE, 2017.

[60] Xilinx. Alveo u280 data center accelerator card data sheet. In *Xilinx*, 2021.

[61] Nvidia. https://www.nvidia.com/content/dam/en-zz/solutions/data-center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf. In *Nvidia*, 2021.

[62] Weiwen Jiang, Edwin H-M Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. Achieving super-linear speedup across multi-fpga for real-time dnn

inference. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–23, 2019.

[63] Nvdla deep learning accelerator. `http://nvdla.org/`. Accessed: 2021-03-25.

[64] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pages 326–331, 2016.

[65] Xinyi Zhang, Weiwen Jiang, and Jingtong Hu. Achieving full parallelism in lstm via a unified accelerator design. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 469–477. IEEE, 2020.

[66] Xinyi Zhang, Yawen Wu, Peipei Zhou, Xulong Tang, and Jingtong Hu. Algorithm-hardware co-design of attention mechanism on fpga devices. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–24, 2021.

[67] Andre Xian Ming Chang and Eugenio Culurciello. Hardware accelerators for recurrent neural networks on fpga. In *2017 IEEE International symposium on circuits and systems (ISCAS)*, pages 1–4. IEEE, 2017.

[68] Shalini Ghosh, Oriol Vinyals, Brian Strope, Scott Roy, Tom Dean, and Larry Heck. Contextual lstm (clstm) models for large scale nlp tasks. *arXiv preprint arXiv:1602.06291*, 2016.

[69] Yequan Wang, Minlie Huang, Xiaoyan Zhu, and Li Zhao. Attention-based lstm for aspect-level sentiment classification. In *Proceedings of the 2016 conference on empirical methods in natural language processing*, pages 606–615, 2016.

[70] Peilu Wang, Yao Qian, Frank K Soong, Lei He, and Hai Zhao. A unified tagging solution: Bidirectional lstm recurrent neural network with word embedding. *arXiv preprint arXiv:1511.00215*, 2015.

[71] Yushi Yao and Zheng Huang. Bi-directional lstm recurrent neural network for chinese word segmentation. In *International Conference on Neural Information Processing*, pages 345–353. Springer, 2016.

[72] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. Data movement aware computation partitioning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 730–744, 2017.

[73] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing the convolution operation to accelerate deep neural networks on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(7):1354–1367, 2018.

[74] Weiwen Jiang, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Lei Yang, et al. Heterogeneous fpga-based cost-optimal design for timing-constrained cnns. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2542–2554, 2018.

[75] Weiwen Jiang, Xinyi Zhang, Edwin H-M Sha, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[76] Xinyi Zhang, Weiwen Jiang, Yiyu Shi, and Jingtong Hu. When neural architecture search meets hardware implementation: from hardware awareness to co-design. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 25–30. IEEE, 2019.

[77] Lei Yang, Zheyu Yan, Meng Li, Hyoukjun Kwon, Liangzhen Lai, Tushar Krishna, Vikas Chandra, Weiwen Jiang, and Yiyu Shi. Co-exploration of neural architectures and heterogeneous asic accelerator designs targeting multiple tasks. *arXiv preprint arXiv:2002.04116*, 2020.

[78] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[79] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[80] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *European Conference on Computer Vision*, pages 213–229. Springer, 2020.

[81] Zhanghao Wu, Zhijian Liu, Ji Lin, Yujun Lin, and Song Han. Lite transformer with long-short range attention. *arXiv preprint arXiv:2004.11886*, 2020.

[82] Robin Cheong and Robel Daniel. transformers.zip: Compressing transformers with pruning and quantization. Technical report, tech. rep., Stanford University, Stanford, California, 2019.

[83] Hanrui Wang et al. Hat: Hardware-aware transformers for efficient natural language processing. *arXiv preprint arXiv:2005.14187*, 2020.

[84] Jonathan Frankle et al. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[85] Haoran You et al. Drawing early-bird tickets: Towards more efficient training of deep networks. *arXiv preprint arXiv:1909.11957*, 2019.

[86] Ari S Morcos, Haonan Yu, Michela Paganini, and Yuandong Tian. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. *arXiv preprint arXiv:1906.02773*, 2019.

[87] Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*, pages 6682–6691. PMLR, 2020.

[88] Jimmy Lei Ba et al. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[89] Fbgemm. Website. `https://github.com/pytorch/FBGEMM`.

[90] Xilinx. Deep learning with int8 optimization on xilinx devices. In *Xilinx WP486*, 2017.

[91] Multi30k. Website. `https://github.com/multi30k/dataset`.

[92] Iwslt. Website. `http://workshop2017.iwslt.org/`.

[93] Daniel Martin, Sandra Malpica, Diego Gutierrez, Belen Masia, and Ana Serrano. Multimodality in vr: A survey. *arXiv preprint arXiv:2101.07906*, 2021.

[94] Hyoukjun Kwon et al. Heterogeneous dataflow accelerators for multi-dnn workloads. *The 27th IEEE International Symposium on High-Performance Computer Architecture (HPCA 2021)*.

[95] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[96] Kenjiro Taura et al. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *Proceedings of HCW*, pages 102–115. IEEE, 2000.

[97] Kaiyuan Guo et al. A survey of fpga-based neural network inference accelerators. *ACM Transactions on TRETS*, 12(1):1–26, 2019.

[98] Chris Riley. Basic tutorial for maximizing memory bandwidth with vitis and xilinx ultrascale+ hbm devices, 2019.

[99] Shifeng Zhang et al. A dataset and benchmark for large-scale multi-modal face anti-spoofing. In *Proceedings of the CVF*, pages 919–928, 2019.

[100] Selvarajah Thuseethan et al. Multimodal deep learning framework for sentiment analysis from text-image web data. In *2020 WI-IAT*. IEEE, 2020.

[101] Tao Shen et al. Facebagnet: Bag-of-local-features model for multi-modal face anti-spoofing. In *Proceedings of the CVF*, 2019.

[102] Xinyu Li et al. Concurrent activity recognition with multimodal cnn-lstm structure. *arXiv preprint arXiv:1702.01638*, 2017.

[103] Jialiang Zhang et al. Improving the performance of opencl-based fpga accelerator for convolutional neural network. In *Proceedings of the 2017 FPGA*, 2017.

[104] Jiantao Qiu et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 FPGA*, pages 26–35, 2016.

[105] Abhinav Podili et al. Fast and efficient implementation of convolutional neural networks on fpga. In *2017 IEEE ASAP*, pages 11–18. IEEE, 2017.

[106] Xuechao Wei et al. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Proceedings of the 54th DAC*, 2017.

[107] Song Han et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 FPGA*, pages 75–84, 2017.