EFFICIENT AI HARDWARE ACCELERATION

BY

XIAOFAN ZHANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Deming Chen, Chair
Professor Wen-Mei Hwu
Professor Sanjay Patel
Assistant Professor Jian Huang
Professor Jinjun Xiong, University at Buffalo

ABSTRACT

The great success of artificial intelligence (AI) has been driven in part by the continuous improvement of deep neural networks (DNNs) with deeper and more sophisticated model structures. DNNs thus become more compute- and memory-intensive. They cause significant challenges for hardware deployment, as they require not only high inference accuracy but satisfied inference speed, throughput, and energy efficiency. Challenges also come from limited hardware resources, restricted power budgets, tedious hardware design programming, intricate hardware verification problems, and time-consuming design space explorations. To address these challenges, this dissertation proposes a comprehensive toolset for efficient AI hardware acceleration targeting various edge and cloud scenarios. It covers the full stack of AI applications, from delivering hardware-efficient DNNs on the algorithm side to building domain-specific hardware accelerators for existing or customized hardware platforms. Major novelties include HLS-based accelerator design and optimization strategies, end-to-end automation tools, and DNN-accelerator co-design methods, which enable highly efficient AI hardware acceleration for various popular AI applications.

Our proposed solution starts with the efficient DNN hardware accelerator design using a High-Level Synthesis (HLS) design flow. Customized hardware accelerators can be developed from a higher abstraction level with a fast response to support emerging AI applications. We demonstrate this method by implementing the first FPGA-based accelerator of the Long-term Recurrent Convolutional Network (LRCN) to enable real-time image captioning. Our design achieves 3.1× higher speedup and 17.5× higher efficiency compared to the optimized GPU-based solution.

We then propose DNNBuilder to improve the efficiency of accelerator design and optimization. It is an end-to-end automation tool providing an integrated design flow from DNN design in deep learning frameworks to board-

level FPGA implementations. Users are no longer required to design and optimize accelerators manually but can enjoy the auto-generated hardware accelerators for desired AI workloads. Novel designs include the fine-grained layer-based pipeline architecture and the column-based cache scheme, which achieve 7.7× and 43× reduction of latency and on-chip memory usage. We demonstrate DNNBuilder by generating state-of-the-art accelerators for various AI services with high quality and energy efficiency.

In addition, we propose a series of efficient design methods to perform algorithm/accelerator co-design and co-optimization, which provides systematic strategies to integrate hardware and software designs. We propose SkyNet, a co-design strategy for hardware-efficient DNN design and deployment with a comprehensive awareness of the hardware constraints. Its effectiveness is demonstrated by outperforming 100+ competitors in the IEEE/ACM Design Automation Conference System Design Contest (DAC-SDC) for low-power real-time object detection. We then extend the co-design to handle two emerging and challenging AI tasks in real-life edge and cloud AI scenarios. We propose F-CAD to deliver customized accelerators for Virtual Reality (VR) applications running on extremely lightweight edge devices. Its generated designs can achieve up to 4.0× higher throughput and up to 62.5% higher energy efficiency than state-of-the-art designs. We propose AutoDistill to address the difficulties of serving large-scale Natural Language Processing (NLP) models in the cloud. Following the co-design strategy, it integrates effective model compression and neural architecture search technologies to deliver high-quality and hardware-efficient NLP pre-trained models. Evaluated on the latest TPU chip, the AutoDistill-generated NLP model can achieve 3.2% higher accuracy and 1.44× faster hardware performance than the state-of-the-art.

These techniques contribute to a new comprehensive toolset that covers hardware accelerator design and optimization on different abstraction levels and DNN-accelerator co-design to deliver efficient AI acceleration for edge and cloud scenarios. It successfully bridges the gap between DNN designs and their hardware deployment and enables easy-accessible, high-quality, and sustainable AI acceleration. As a result, we are able to demonstrate state-of-the-art solutions for various popular and emerging AI applications.

To my parents, for their love and support.

ACKNOWLEDGMENTS

My Ph.D. journey is a precious and unforgettable chapter in my life. I am exceptionally grateful for receiving a lot of help and support from my advisors, colleagues, friends, and family.

First, I would like to express my sincere gratitude to my advisor, Prof. Deming Chen, for trusting and inviting me to join the ES-CAD research group. I have learned a lot from his creativity, dedication, and passion for research, which have shaped my personality as a passionate researcher. Without his guidance, it would not be possible for me to get my achievements, including this dissertation. I would like to thank Prof. Wen-mei Hwu and Prof. Jinjun Xiong, my mentors and closest collaborators in the C3SR research center, for their continuous support, guidance, and encouragement. They are absolutely the role models of great researchers with immense knowledge, brilliant ideas, and ceaseless enthusiasm. I am grateful to have Prof. Sanjay Patel and Prof. Jian Huang on my thesis committee. They have provided valuable feedback and suggestions for improving my dissertation.

Besides, I would like to thank all my labmates in the ES-CAD research group for sharing with me their brilliant ideas and exciting discussions. Thanks to Xinheng Liu, Anand Ramachandran, and Chuanhao Zhuge who worked closely with me to build the first LRCN accelerator and prepared for the prototype in Chapter 3. Thanks to Cong Hao, Yuhong Li, Sitao Huang for working toward the DNN/accelerator co-design idea in Chapter 5 and their great efforts to win the DAC-SDC. I want to thank Di He, Wei Zuo, Qin Li, Yao Chen, Hanchen Ye, Yuan Ma, Junhao Pan, Zehua Yuan, for being awesome labmates. I enjoy working with them on numerous exciting projects. I also want to thank my friends in the Coordinate Science Laboratory (CSL) who always encouraged me to think outside the box and motivated me to work harder.

Next, I would like to thank my collaborators outside the University of Illi-

nois, who gave me insights and helped me complete this dissertation. It is great fun and rewarding to work with them. I would like to thank Yonghua Lin, Junsong Wang, Chao Zhu, and Yubo Li at IBM Research. Their pioneering works on end-to-end automation tools led me to the DNNBuilder project in Chapter 4. Many thanks for their insights, discussions, and contributions. I would like to thank Yuecheng Li, Dawei Wang, Pierce Chuang, and Shugao Ma at Meta Reality Labs Research. They helped me identify the bottlenecks of current VR accelerators and made the F-CAD design possible in Chapter 6. I also want to thank Emma Wang and Zongwei Zhou at Google for guiding me toward the research topics on large-scale model serving in the cloud and for their great support in the AutoDistill project presented in Chapter 7.

Finally, I would like to thank my fiancée and my parents for their unconditional love and support. Words cannot express my appreciation, and I know that they are always proud of me no matter what I achieve.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS ix

CHAPTER 1 INTRODUCTION 1
1.1 Motivations 2
1.2 Contributions 4

CHAPTER 2 BACKGROUND AND RELATED WORK 8
2.1 Deep Neural Network 8
2.2 Hardware-efficient DNN designs 12
2.3 Efficient DNN accelerators 14
2.4 Efficient Co-design Strategies 15

CHAPTER 3 HLS-BASED DESIGN AND OPTIMIZATION STRATEGIES FOR ACCELERATING LRCN 16
3.1 Introduction 16
3.2 Design Challenges of Accelerating Video Analysis 17
3.3 Design Methodologies for Building LRCN Accelerator 19
3.4 Hardware Implementation and Comparison 25
3.5 Conclusion 27

CHAPTER 4 DNNBUILDER: AN AUTOMATED TOOL FOR BUILDING HIGH-PERFORMANCE DNN HARDWARE ACCELERATORS FOR FPGAS 29
4.1 Introduction 29
4.2 The Proposed Automation Flow 30
4.3 Accelerator Architecture 32
4.4 Automatic Resource Allocation 37
4.5 Experimental Results 41
4.6 Conclusion 49

CHAPTER 5 SKYNET: EFFICIENT DNN-ACCELERATOR CO-DESIGN STRATEGIES 51
5.1 Introduction 51
5.2 DNN-Accelerator Co-Design 52
5.3 A Bottom-up DNN Design strategy 55

5.4 The SkyNet . . . 59
5.5 A Top-down Accelerator Design Strategy . . . 60
5.6 Experimental Results on DAC-SDC . . . 62
5.7 SkyNet Extension on GOT-10K Object Tracking . . . 69
5.8 Conclusion . . . 71

CHAPTER 6 F-CAD: A CUSTOMIZED ACCELERATOR DESIGN FLOW FOR EMERGING EDGE VR APPLICATIONS . . . 72
6.1 Introduction . . . 72
6.2 Design Challenges of accelerating VR codec avatar . . . 74
6.3 The Proposed F-CAD Design Flow . . . 77
6.4 Accelerator Architecture . . . 79
6.5 Multi-Branch Design Space Exploration . . . 81
6.6 Experimental Results . . . 86
6.7 Conclusion . . . 89

CHAPTER 7 AUTODISTILL: AN END-TO-END FRAMEWORK TO EXPLORE AND DISTILL HARDWARE-EFFICIENT LANGUAGE MODELS . . . 90
7.1 Introduction . . . 90
7.2 Knowledge Distillation Background and Challenges . . . 91
7.3 The Proposed AutoDistill . . . 93
7.4 Flash Distillation . . . 96
7.5 Hardware-Aware Model Selection . . . 99
7.6 Experimental Results . . . 101
7.7 Conclusion . . . 112

CHAPTER 8 CONCLUSION . . . 114

REFERENCES . . . 117

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
ACM	Association for Computing Machinery
ASIC	Application-Specific Integrated Circuit
BN	Batch Normalization
BO	Bayesian Optimization
BRAM	Block Random Access Memory
CE	Compute Engine
CPU	Central Processing Unit
CNN	Convolutional Neural Network
Conv layer	Convolutional layer
CPF	Channel Parallelism Factor
CTC	Computation-to-Communication
DAC-SDC	Design Automation Conference System Design Contest
DNAS	Differentiable Neural Architecture Search
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
DSE	Design Space Exploration
FC layer	Fully-Connected layer
FF	Feed-Forward
FFT	Fast Fourier transform

FIFO	First In First Out
FM	Feature Map
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
GFLOP	Giga Floating-point Operations
GLUE	General Language Understanding Evaluation
GOPS	Giga Operations
GPU	Graphics Processing Unit
HD	High-Definition
HDL	Hardware Description Languages
HLS	High-Level Synthesis
HMD	Head Mounted Display
IEEE	Institute of Electrical and Electronics Engineers
IOU	Intersection Over Union
IP	Intellectual Property
KPF	Kernel Parallelism Factor
LRCN	Long-term Recurrent Convolutional Network
LSTM	Long Short-Term Memory
MHA	Multi-Head Attention
MLM	Masked Language Modeling
NAS	Neural Architecture Search
NLP	Natural Language Processing
NSP	Next Sentence Prediction
PC	Personal Computer
PE	Process Element
PSO	Particle Swarm Optimization
QPI	QuickPath Interconnect

RAM	Random Access Memory
RNN	Recurrent Neural Network
RTL	Register-Transfer Level
RX	Receiver
SoC	system-on-Chip
SQuAD	Stanford Question Answering Dataset
TPU	Tensor Processing Unit
TX	Transmitter
VR	Virtual Reality

CHAPTER 1

INTRODUCTION

The recent development of DNNs has made AI more relevant and accessible to the general public. We have seen that most DNN technologies have been deeply integrated into applications related to our daily lives, such as image recognition, object tracking, speech recognition, language translation, self-driving cars, and augmented and virtual reality. These applications can be easily found running at the edge devices or in the cloud servers, requiring high inference accuracy to achieve intelligent responses and aggressive inference speed, throughput, and energy efficiency to meet various real-life demands.

As DNNs become more complicated, developing and serving the DNN-enabled applications requires more compute and memory resources, longer latency, and greater energy consumption. For example, the amount of computation for DNN training has risen by over 300,000× in just six years between AlexNet [1], the champion DNN model of 2012 ImageNet competition, and AlphaGo Zero [2], the AI player for the board game Go with superhuman skills [3]. For DNN inference, by taking image recognition as an example, we have seen 16× model complexity increase from AlexNet to ResNet-152 [4]. By processing the same size of input images from the ImageNet dataset, AlexNet requires 1.4 Giga floating-point operations (GFLOP) and achieves around 85% top-5 accuracy, while ResNet-152 takes 22.6 GFLOP and reaches around 95% top-5 accuracy.

Such exponentially increasing compute and memory demands cause great difficulties for DNN deployment on hardware. Fortunately, we have seen continuous progress published in recent studies to build domain-specific accelerators for efficiently handling the resource-demanding DNN workloads. These accelerators attempt to take advantage of different hardware design styles, such as adopting acceleration libraries on CPU [5], exploring kernel optimization on GPUs [6, 7], building customized accelerator on FPGAs [8, 9, 10] and ASICs [11, 12, 13], to improve the speed and efficiency of DNN

inference and training processes. Among these accelerator designs, FPGA- and ASIC-based designs can be fully customized to implement the neural network functionality with improved latency, throughput, and energy consumption compared to CPU- and GPU-based designs.

However, developing customized DNN accelerators still presents significant challenges, such as the tedious programming using hardware description languages (HDL), the intricate hardware verification problems, and the time-consuming design space exploration process. All these challenges seriously reduce the accelerator design efficiency and easily cause sub-optimal solutions given available hardware resources. In addition, emerging AI applications usually adopt DNNs with diverse layers and deeper network structures. They often have multiple requirements, such as high-definition image/video input support, real-time capability, flexible batch-process configuration, etc. It is still unclear how to effectively and systematically exploit the available resources and produce customized DNN accelerators that meet all the desired requirements.

This dissertation provides a comprehensive toolset to address the challenges mentioned above and deliver efficient AI hardware acceleration for various real-life scenarios. Our proposed toolset includes HLS-based DNN accelerator design and optimization strategies, end-to-end automation tools for customized accelerator designs, and DNN-accelerator co-design and co-optimization strategies. We demonstrate the proposed designs by generating state-of-the-art solutions to accelerate popular AI applications, including video analysis in Chapter 3, image classification in Chapter 4, object detection and tracking in Chapter 5, VR codec avatar in Chapter 6, and popular NLP tasks in Chapter 7.

1.1 Motivations

1.1.1 HLS-based design and optimization strategies

HLS allows the use of high-level (behavioral-level) programming languages, such as C/C++, to be the abstract descriptions of hardware functions, which significantly improve the design efficiency compared to the conventional development using HDL on register-transfer level (RTL). A higher-level de-

scription requires fewer codes and gives faster simulation. In an example mentioned in [14], a hardware design with one million logic-gates-circuit requires roughly 300K lines of RTL codes, but it can be described by only 40K lines of C codes, where 7× code size reduction is achieved. In addition, HLS provides a variety of loop-based optimization strategies, such as loop unrolling and loop pipelining, which exhibit great potential for handling DNN layers, as most of the layers can be expressed in the form of intertwined loops. In our proposed methods, we leverage the advantage of HLS design flow as well as various optimization strategies and eventually deliver accelerators with improved hardware performance and efficiency. We choose FPGAs as the targeted devices because they can be fully customized. It is the critical feature we need in this dissertation, but CPUs and GPUs cannot provide it. Compared to ASICs, FPGAs provide a higher degree of design flexibility, faster time-to-market, and less non-recurring engineering cost.

1.1.2 End-to-end automation tools

To step further in the direction of improving accelerator design efficiency, we propose end-to-end automation tools to seamlessly bridge the gap between fast DNN construction in software (the machine learning framework, e.g., Caffe [15], TensorFlow[16], PyTorch [17], etc.) and slow hardware implementation. These proposed tools support inputs described on a much higher abstraction level, which are the same as DNN definition files used in the machine learning frameworks. So, DNNs can be directly imported to the proposed tools, and no manual code conversions are required. The benefits of proposing end-to-end tools are not only to support higher abstract DNN descriptions, but also integrate design space exploration (DSE) engines for effective and systematical explorations and deliver highly optimized accelerators to meet specific performance targets with the available resource budgets.

1.1.3 DNN-accelerator co-design strategies

Although hardware accelerators can be fully optimized to deliver improved performance, there are still challenges from the conflict between limited avail-

able hardware resources and increasingly demanding application-specific requirements. To address these conflicts, optimizations need to be done on both software and hardware sides to compress the DNNs and make them hardware-efficient, as well as to build powerful hardware accelerators by fully utilizing the available resource budgets. However, conventional approaches still construct DNN and accelerator in two separate steps and perform optimization iteratively for improved software and hardware metrics. It is difficult to balance the DNN design metrics (e.g., accuracy, model complexity, robustness, etc.) and hardware accelerator design metrics (e.g., throughput, latency, resource usage, power consumption, etc.) due to the inefficient coordination between hardware and software, resulting in a time-consuming and error-prone procedure with no guarantee of convergence. To address this issue, we propose DNN-accelerator co-design strategies to consider software and hardware metrics simultaneously. DNNs are designed to satisfy accuracy demands and must be aware of the hardware constraints with rational network configurations. At the same time, the accelerators need to provide extensive support for different DNN components without introducing too many restrictions on network design and guarantee performance to meet the specifications. To achieve this goal, we introduce a unified intermediate representation for both DNNs and hardware accelerators to explore the joint optimization opportunities and deliver better solutions than optimizing DNNs or accelerators alone.

1.2 Contributions

In this dissertation, we provide a comprehensive toolset to cover hardware accelerator design and optimization on different abstraction levels and DNN-accelerator co-design to deliver efficient AI acceleration for edge and cloud scenarios. Compared to previously published solutions, our proposed toolset provides more comprehensive support to enable AI hardware acceleration from the behavioral level while working with DNN-accelerator co-design. It forms a comprehensive design space of the customized hardware accelerators as well as the DNN models and provides various optimization strategies to create the final optimized design given multiple constraints, such as user-specific requirements and available hardware resources. The proposed solu-

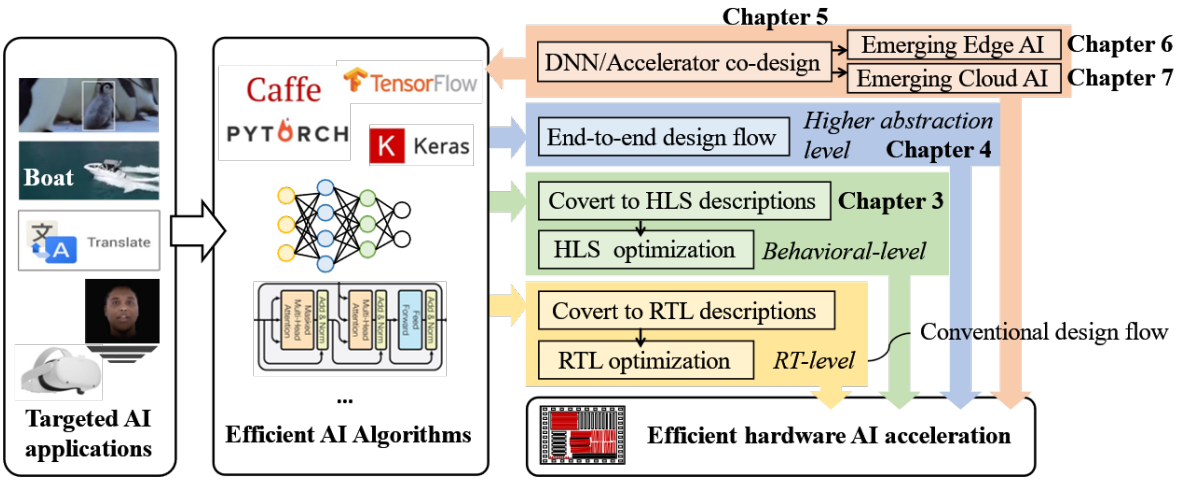

Figure 1.1: A comprehensive toolset is proposed to address the challenges of AI hardware acceleration. The contributions in this dissertation include HLS-based accelerator design and optimization strategies [18] in Chapter 3, end-to-end automation tools [10] in Chapter 4, DNN-accelerator co-design methods [19] in Chapter 5, and demonstrations of the proposed designs on emerging edge VR [20] and cloud NLP applications [21] in Chapter 6 and 7.

tions have been demonstrated in various popular AI applications and have led to multiple breakthroughs in delivering efficient AI hardware acceleration.

As shown in Figure 1.1, our work focuses on bridging the gap between DNN designs and their hardware deployment as well as enabling DNN/accelerator co-design and co-optimization. This figure first shows the conventional design flow: DNNs are developed on machine learning frameworks and converted to RTL descriptions for hardware deployment. Our work provides more efficient DNN mapping solutions using higher abstraction levels, such as the HLS-based design methods in Chapter 3 and the end-to-end solutions in Chapter 4. We also contribute to the DNN/acceleration co-design and co-optimization (Chapter 5 $\sim$ 7) to enable joint-optimization opportunities between software and hardware. Detailed contributions are summarized as follows:

- HLS-based design and optimization methods for accelerating complicated video analysis DNN on FPGAs [18]. Our proposed toolset provides the **first** FPGA implementation of LRCN, a hybrid DNN model with both Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN). It demonstrates the feasibility of using HLS to implement high-performance DNN accelerators and overcomes the design challenges in managing computational complexity, on-chip mem-

ory limitation, and external memory bottleneck. We then release the **first** open-source LRCN accelerator design, which provides successful design templates to address challenges in managing compute- and communication-intensive AI workloads. We will introduce the detailed designs in Chapter 3.

- An end-to-end design automation tool called DNNBuilder [10]. We further provide an integrated design flow from DNN models in deep learning frameworks to board-level FPGA implementations. DNNBuilder further improves the hardware accelerator design efficiency as it automates the customized DNN accelerator design process and overcomes the significant challenges encountered before, such as hardware programming, hardware verification, and design space exploration. DNNBuilder gives a new direction for accelerator architecture design by proposing a fine-grained layer-based pipeline and a column-based cache scheme. These novel technologies reduce 7.7× latency and 43.0× on-chip memory consumption. Because of its novel contributions, DNNBuilder has been awarded the **IEEE/ACM William J. Mccalla ICCAD Best Paper Award**. It is also an open-source project, which has been adopted by the industry. We will introduce this work in Chapter 4.

- DNN-accelerator co-design strategies to integrate hardware and software designs. In addition to the methods mentioned above, we expand our proposed toolset to support DNN-accelerator co-design and provide more effective methods for building optimized AI solutions considering larger and more challenging co-design spaces. The proposed co-design strategies help deliver efficient AI acceleration with careful considerations of both DNN design diversity and hardware constraints. Particularly, we propose a uniform intermediate representation called Bundle to connect software and hardware metrics. These strategies are demonstrated in SkyNet to provide efficient and real-time object detection and tracking [19]. SkyNet won the **First Place Winner Awards** of the DAC-SDC by generating solutions that significantly outperformed other 100+ competitors. We will introduce SkyNet in Chapter 5.

- To cover emerging edge AI applications, we propose F-CAD, the **first**

design flow for accelerating emerging VR applications running on lightweight edge devices [20]. F-CAD introduces a novel elastic accelerator architecture and a dynamic design space exploration to optimize hardware designs. It achieves up to 4.0× higher throughput and up to 62.5% higher energy efficiency than state-of-the-art designs and can perfectly meet the demanding VR requirements. It is a timely addition to our toolset to overcome the emerging challenges. We will introduce F-CAD in Chapter 6.

- Finally, we integrate AutoDistill into our toolset to support emerging cloud AI applications. AutoDistill is the **first** fully automated framework that integrates model distillation and neural architecture search (NAS) to deliver hardware-efficient NLP pre-trained models for cloud-based AI services [21]. It comprehensively considers both model quality and serving latency on the targeted cloud server following the co-design strategies. Compared to the state-of-the-art compressed BERT model, AutoDistill provides better solutions with 1.44× faster hardware performance and 3.2% higher accuracy. We will introduce AutoDistill in Chapter 7.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter will briefly introduce DNNs and popular solutions to make DNNs efficient for hardware deployment. We will also cover the design methodologies and recent development of AI accelerator designs. At last, we will introduce recent software/hardware co-design and co-optimization methods to implement efficient AI solutions for real-life applications.

2.1 Deep Neural Network

AI is a broad concept that refers to the intelligence provided by smart machines with the capability of performing tasks that typically require human intelligence. There are many AI applications in our daily lives, such as video analysis in a security camera to detect crowd density [22, 23], language translation in a cell phone to ensure communication when traveling abroad [24, 25, 26], a recommendation system to achieve personalized advertising during internet surfing [27, 28], immersive communication using VR technologies [29, 30, 31], etc. AI can be achieved in different ways, and, among all, deep learning is one of the most popular approaches, which processes inputs through DNNs. With the recent breakthroughs in DNNs and improvements in computing power and datasets, AI applications have become increasingly diverse and accessible, continually reshaping our lives in many different ways [32]. In this section, we will introduce the DNNs mentioned in the following chapters, which include AlexNet [1] and VGG [33] for image classification, LRCN for video analysis [23], and a multi-branch DNN for VR codec avatar [29].

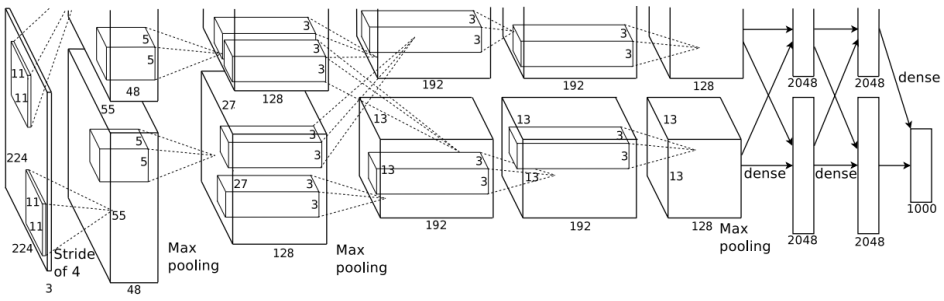

Figure 2.1: AlexNet architecture [1].

Figure 2.2: VGG architecture [33].

2.1.1 CNN architectures

A CNN includes multiple intermediate layers between the input and output layers, and each intermediate layer consists of artificial neurons for transforming the input information following the predefined network connections. It is one of the most widely used deep learning algorithms for computer vision applications, which takes images or video frames as inputs and then recognizes the spatial information of the input contents. A typical CNN is structured as a series of layers, including Conv layer, pooling layer, activation layer, and FC layer.

By taking AlexNet [1] as an example (shown in Figure 2.1), its major layers include five Conv layers using three kernel sizes (11×11, 5×5, and 3×3), three pooling layers, and three FC layers. Compared to the prior designs (e.g., LeNet), AlexNet is very deep and complicated, and it achieves 57.2% top-1 accuracy on the ImageNet dataset. Another example, VGG-16, is shown in Figure 2.2, which is deeper with 13 Conv layers and three FC layers. It makes the improvement over AlexNet by replacing large kernels with the 3×3 kernels. It demonstrates that multiple stacked smaller size kernel with a deeper network structure is better than the one with a larger size kernel but shallow network. Its ImageNet top-1 accuracy is 68.5%. We

will present the customized hardware accelerator designs for handling these CNNs in Chapter 4.

2.1.2 LRCN for video content analysis

When dealing with complex AI applications, different types of DNNs can be combined together to maximize their different strengths. For example, a DNN called LRCN adopts both CNNs and RNNs to perform activity recognition and content captioning based on the input videos [23]. A typical implementation of the LRCN includes an AlexNet [1] for the CNN and multiple LSTM layers [34] for the RNN. Its workflow is shown in Figure 2.3, where input video frames are first passed to a CNN to extract its spatial features and these features are then fed into an RNN composed of multiple long short-term memory (LSTM) layers to finally produce a video content description. The advantages of LRCN come from its hybrid neural network structure combining both CNN and RNN layers: the CNN is first used to capture the input's spatial information, while the following RNN takes these spatial features sequentially and generates text descriptions.

Although LRCN is a powerful tool for video analysis, it involves more complex network structures and requires more intensive compute and memory capacity during inference compared to a single CNN or RNN. There is no doubt that we need to use hardware acceleration to meet various performance requirements. In Chapter 3, we will introduce HLS-based design and optimization strategies for building a hardware accelerator for efficiently running LRCN.

2.1.3 Multi-branch DNN for VR codec avatar

Another DNN example comes from an emerging AI application called VR codec avatar [29]. It performs as a decoder to render and display the codec avatar in the receiver's VR headset with the overall workflow shown in Figure 2.4. The goal of this DNN is to generate photo-realistic and three-dimensional reproductions of human appearances and real-time expressions, so that we can achieve VR telepresence with more effective communications. The state-of-the-art codec avatar decoder contains at least three branches for generating

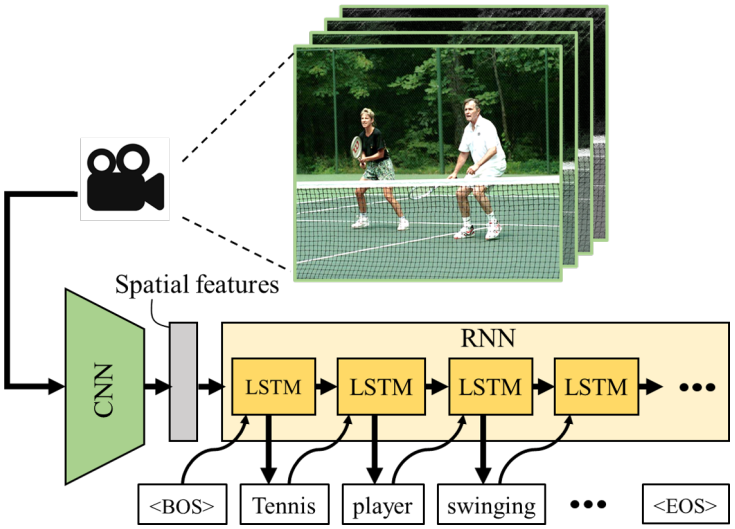

Figure 2.3: LRCN adopts both CNN and RNN for video content analysis. Input video frames are first processed by the CNN to extract the spatial features, and these features are then passed to the RNN for generating descriptions. The CNN generates spatial features of the inputs while the RNN outputs a sentence as the video content description in a recurrent manner, which starts with the entry <BOS> (beginning of sequence) and ends up with <EOS> (end of sequence) [35].

facial geometry (3D vertices), UV texture (a 2D surface of a 3D model following U- and V-axis), and warp field (specular effects), of which the second and third branches have a common front part. The decoder can contain more than 13.6 GFLOP and 7.2 million parameters. Unlike general DNNs, the decoder introduces complex data dependencies by adopting multi-branch structures and high-definition intermediate results for high-quality VR avatar textures. Other features come from the customized convolutional (Conv) layer, where each output pixel has its dedicated bias (also named untied bias for improved image output quality) instead of sharing one bias across pixels within the same output channel.

The unique multi-branch feature and customized layers from codec avatar decoders cause complicated dataflows and high compute and memory demands during inference, which make existing DNN accelerators ineffective. Challenges include the enormous and unevenly distributed computations and the substantial memory footprints. It becomes even more challenging for hardware accelerators with limited resources aiming at real-time response with high throughput performance. Chapter 6 will introduce F-CAD to ad-

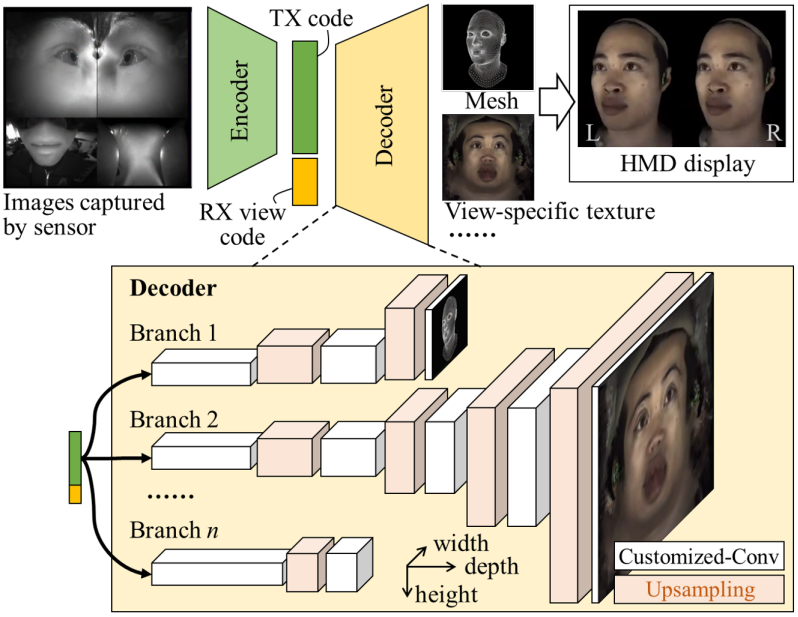

Figure 2.4: The workflow of VR codec avatar: from images captured by the transmitter (TX) to VR avatar displayed by the receiver's (RX) head mounted display (HMD). The decoder is the most complicated part with a multi-branch DNN to generate different components of the avatar (e.g., mesh vertices in Branch 1, the view-specific texture in Branch 2, etc.) [20].

dress these challenges and deliver hardware acceleration for codec avatar decoders.

2.2 Hardware-efficient DNN designs

In general, a DNN contains millions of parameters and requires billions of operations during inference. To successfully deploy DNNs onto hardware with desired performance, researchers and developers start focusing on network compression to reduce the network complexities and lower the compute and memory demands. Recent research has demonstrated the possibility of using lower bit-width data to represent original floating-point parameters, such as using 16/8-bit quantization or even binary and ternary data representation for network parameters or intermediate results [13, 36, 37, 38, 39, 40]. These solutions are intended to replace the hardware-intensive floating-point multiplications by fixed-point operations or logical operations, so that DNNs can be more efficient when deploying on hardware platforms.

Another method to compress DNN is known as network pruning, which aims to reduce the redundancy of DNN structures [41, 42, 43]. According to the published pruning strategies, the relatively less important connections between DNN layers are discarded, and network retraining is then performed to regain accuracy. Significant reductions can be achieved on the classic DNNs, such as AlexNet [1] and VGG-16 [33]. Since the major benefit of network compression comes from the fully-connected (FC) layers, to continuously have effective pruning results for latter DNNs (e.g., GoogleNet [44] and ResNet [4]) with reduced FC layers, more sophisticated algorithms are required to be integrated in network pruning, such as using evolutionary algorithms [45], alternating direction method of multipliers [46], and iterative pruning [47].

As most of the computations happen inside the Conv layers, previous works also attempt to reduce the computation complexity by using depth-wise separable Conv layers for image classification and ubiquitous keyword-spotting applications [48, 49]. The depth-wise separable structure can effectively reduce the number of operations and provide more compact DNN designs for resource-constrained hardware. To further improve the DNN deployment on hardware, layer fusion is proposed in [50] to minimize data movements between on-chip and off-chip memory.

More recently, large-scale pre-trained language models, such as GPT-3 [51] and BERT [52], have achieved state-of-the-art results on many NLP tasks. So, serving these models has become essential in cloud servers. With the advent of such large-scale language models, minimizing the serving cost is becoming increasingly important as the cost of serving adds up from every request. To reduce the model size, knowledge distillation is adopted to compress large models by generating a compact model (the student model) and training it based on the trained pattern of a larger model (the teacher model) [53]. Earlier work has focused on distilling large models, like BERT, to task-specific compact designs with less redundancy in model architecture [54, 55, 56], or task-agnostic pre-trained models, which can then be fine-tuned to different downstream tasks [57, 58].

2.3 Efficient DNN accelerators

To support the complicated DNN workloads, we have seen extensive studies of building domain-specific hardware accelerators. These accelerators attempt to take advantage of customized or specialized hardware and software designs, such as adopting acceleration libraries on CPUs [5], exploring kernel optimization on GPUs [7], and building customized accelerators on FPGAs [8, 9, 10] and ASICs [11, 12, 13] to improve the speed and efficiency of DNN inference and training processes.

Among these designs, FPGA-based solutions have been rapidly developed and become one of the most promising solutions to provide improved performance and efficiency [59, 10, 60, 61]. Designs presented in [8] explore the design space of loop optimizations in a CNN implementation to locate the best implementation point. To improve the hardware efficiency, the authors in [9] investigate dynamic quantization schemes for quantizing both DNN parameters and intermediate feature maps. The designs in [62, 37] support binary and ternary quantization, which further relax the intensive computational pressure using logical operations. Researchers also focus on building accelerators for sparse matrix-vector/matrix multiplication and attempt to address the irregular computation and memory accesses [63, 64]. These accelerators can be applied to DNN workloads for more efficient executions with a high sparsity level where zero elements are frequently involved [65]. In this dissertation, we adopt network pruning to eliminate network redundancy and treat them as dense DNN workloads. We will continue investigating the sparse matrix optimization for DNNs in our future work. Other optimizations include implementing fast Conv algorithms, such as using Winograd-based solutions and Fast Fourier transform (FFT) to replace the original spatial Conv operations [66, 67]. In [68], the targeted DNN is first compressed and then deployed onto FPGA to achieve higher efficiency.

Still, developing customized accelerators presents significant challenges, such as the tedious hardware design process, the intricate hardware verification problems, and the time-consuming design space exploration during DNN deployment. To alleviate these challenges, recent investigations have started focusing on techniques including high-level synthesis [18, 69, 70] and end-to-end design frameworks for fast DNN accelerator design and efficient workload deployment [71, 10, 72, 73, 35]. They support high abstraction inputs,

such as Python-based DNN descriptions used by popular machine learning frameworks (e.g., Caffe [15], TensorFlow [16], PyTorch [17]), so DNNs can be directly imported without manual code conversions and be parsed and then mapped onto hardware. In Chapter 4, we will introduce DNNBuilder [10] to provide an end-to-end design flow for building high-performance and efficient DNN accelerators. DNNBuilder introduces a fine-grained pipeline structure, a novel caching scheme between pipeline stages, and highly optimized RTL network layers with arbitrary quantizations to deliver high throughput, low latency, and desired network accuracy.

2.4 Efficient Co-design Strategies

Recent research also focuses on co-design and co-optimization opportunities for both DNNs and hardware accelerators in order to achieve more efficient DNN hardware deployment [61, 74, 75, 76, 77, 35, 78]. These co-design approaches have been studied with remarkable achievements by combining multiple optimization techniques across DNN and accelerator designs. For example, while NAS has been largely successful in designing high-quality DNN models [79, 80], it is extended to consider hardware metrics to deliver DNNs with high inference accuracy and hardware efficiency when deploying onto targeted devices [81, 82, 83]. Recent work also shows increasing interest to incorporate hardware performance feedback with network compression technologies, such as knowledge distillation, to better serve the large-scale NLP models in the cloud [57, 21]. Driven by the success of such a co-design strategy, other types of co-design methods are also proposed recently. They include software/compiler co-design [84, 85], compiler/hardware co-design [86], software/system co-design [77], etc.

In Chapter 5, we will introduce one of the pioneer co-design solutions called SkyNet to deliver hardware-efficient AI solutions. SkyNet is demonstrated by winning the competitive DAC-SDC for providing the best solution in low power object detection [19]. In Chapter 6 and 7, we will present co-design strategies called F-CAD [20] and AutoDistill [21] to handle emerging edge VR and cloud NLP applications.

CHAPTER 3

HLS-BASED DESIGN AND OPTIMIZATION STRATEGIES FOR ACCELERATING LRCN

3.1 Introduction

DNN-based video analysis has become one of the most essential and challenging tasks to capture implicit information from video streams. It allows machines to handle more real-life applications that originally require human efforts. For example, videos captured by surveillance systems can be fast exterminated by machines to identify particular dangerous scenes without manual interventions. Some emerging applications, such as video and image annotations, can also benefit from the development of DNN-based video analysis in order to generate the desired output descriptions automatically.

Among various DNN designs, LRCN presents the most promising capability for video analysis, such as performing activity recognition and content captioning for the input videos [23]. As introduced in Section 2.1, LRCN contains two types of DNNs: an AlexNet [1] for the CNN and multiple LSTM layers [34] for the RNN. In this case, LRCN contains 2.22 GFLOP and 86.56 million parameters for processing one input video frame. Such a unique DNN combination exhibits significantly different layer characteristics with respect to the computational complexity (the number of operations required in one layer) and memory demands (the amount of data fetched and stored in a certain period). For example, the Conv layers in CNN are computationally intensive, so accelerating these layers is often limited by the available compute units. In contrast, the FC layers are highly memory-intensive, where on-chip memory size and the external memory access bandwidth dominate their achievable performance.

With such characteristics, using general-purpose processors may fail to deliver satisfactory performance and efficiency for video analysis using LRCN. In this chapter, we will adopt the HLS design flow and build a customized

hardware accelerator for running LRCN inference. The key contributions are summarized as follows:

- We present a complete HLS design flow from C-level descriptions to FPGA board-level implementations for developing a video analysis system. We propose highly optimized HLS IPs (e.g., Conv, Pooling, and LSTM IPs) to leverage DNN implementations and essential supporting modules, such as the memory hierarchy and the data streaming module, to ensure the efficient operation of the proposed accelerator.
- We introduce a resource partitioning solution called REALM to provide layer-wise resource allocation guidelines to configure the proposed HLS IPs for minimum overall latency.
- We demonstrate the feasibility of using the proposed HLS design and optimization strategies to deliver a higher-performance LRCN accelerator. By implementing our proposed accelerator on the Xilinx VC709 FPGA board, we achieve 3.1× higher speedup and 17.5× higher efficiency than an optimized GPU-based design.

3.2 Design Challenges of Accelerating Video Analysis

Before introducing the detailed approaches for building the accelerators, in this section, we summarize three design challenges from DNN structure, real-life requirements, and hardware implementation.

3.2.1 Diverse DNN layers

The unique combination of CNN and RNN creates unbalanced demands for different types of layers in LRCN. These layers exhibit different characteristics in consuming computation and memory resources. Regarding an LRCN design that contains an Alexnet as the front-end CNN and 15 iterations of LSTM layers as the back-end RNN, processing one video frame costs 2.22 GFLOP and consumes 86 million DNN parameters occupying 346 MB of memory.

Table 3.1: Percentage of computation and memory consumption in LRCN

Layer	Compute resource	Memory resource	CTC ratio
Conv	60.06%	2.69%	1700.9
FC	5.29%	67.73%	2.0
LSTM	34.65%	29.58%	30.1

As shown in Table 3.1, we break down the resource demands for three major types of layers as Conv and FC layers in the CNN and LSTM layers in the RNN. The computation demand is dominated by the Conv layers with 60.06% of the total computation, while the memory consumption mostly comes from the FC layers, peaking at 67.73% over all required parameters. Since DNN parameters can not be fully accommodated on chip, most of them are stored in the external memory and are only fetched if necessary by consuming external memory access bandwidth. Therefore, we adopt computation-to-communication (CTC) ratio to indicate the layer-wise data reuse behaviors. Layers with higher CTC ratios mean that they are expected to be compute-bounded, while layers with lower ratios indicate that they are memory-bounded. Such diverse layer characteristics pose tough challenges for efficient hardware accelerator design and require the proposed design to have great adaptability for different layers with expected performance.

3.2.2 Real-life application requirements

By targeting video analysis applications, accelerators are required to handle streaming inputs and deliver satisfactory throughput performance to match the video frame rate. Techniques, such as using a large batch size to increase the degree of computing parallelism and data reuse, are thus often carried out. However, this type of application also needs to support frequent user interactions, making real-time response indispensable. The design difficulty then lies in the place where the proposed hardware accelerator needs to deliver high throughput without using large batch sizes since the extra delay in collecting batch inputs may fail to meet the real-time requirement.

3.2.3 Hardware implementation difficulties

Major layers, which dominate the compute and memory resources – such as Conv, FC, and LSTM layers, are constructed by multiple nested loops from the algorithmic perspective. By following the HLS design flow, developers can adopt the loop optimization strategies (e.g., loop unrolling and pipelining) provided by HLS directives to create hardware instances with improved performance. However, inserting directives may not guarantee better performance since some specific loop structures can invalidate certain directives and thus result in undesired designs. For instance, the loop unrolling directive should not be applied to the loops with complex data dependency. In addition, an optimal hardware implementation always requires careful resource allocation since the DNN layers vary significantly in compute and memory demands. If a DNN layer is memory-bounded, its achievable performance is less affected by the allocated computation resources but by how frequently it accesses the memory and whether enough memory access bandwidth can be provided.

3.3 Design Methodologies for Building LRCN Accelerator

To overcome the challenges mentioned in Section 3.2, we introduce the design methodologies that we used for developing the LRCN accelerator with an HLS design flow. The key techniques include 1) highly configurable HLS IPs for designing Conv, FC, and RNN layers for building all network layers of the LRCN; 2) a resource partitioning solution that provides guidelines for resource allocation per layer for minimum overall latency; 3) network compression technologies, including network pruning and dynamic quantization, to ensure our LRCN design can exploit the strength of FPGA platforms, and 4) a hierarchical memory subsystem design for hiding data fetch latency.

3.3.1 HLS IP for LRCN

Our IP-based design methodology provides opportunities to quickly implement a high-quality FPGA design and gives an efficient approach for design

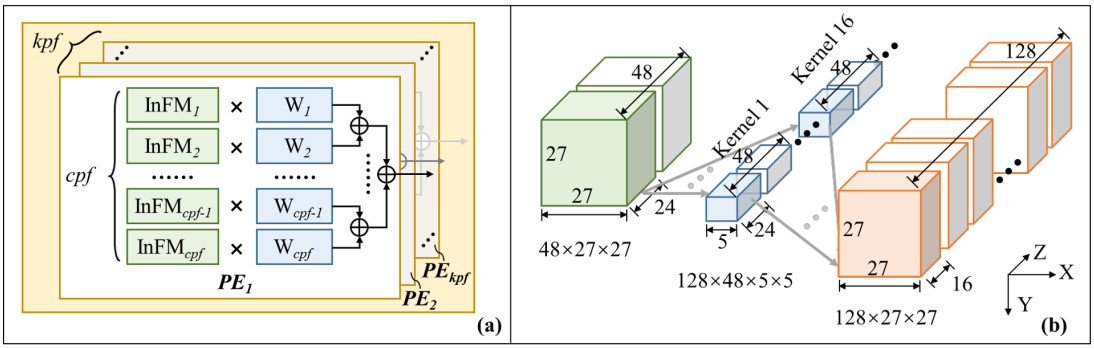

Figure 3.1: (a) The proposed compute engine (CE) design and (b) an example of how the proposed CE leverage a Conv layer in LRCN.

space exploration by adjusting different IP configurations. To leverage its benefits, the proposed HLS IP covers the most critical and universal operations in DNNs, the multiply-accumulation, and its parametric feature makes sure it can adapt to implement different layers of LRCN.

We propose a compute engine (CE) for handling computations in every DNN layer. As shown in Figure 3.1 (a), CE is represented as a yellow rectangle, which provides two-dimension parallel processing along with input and output channels of the targeted DNN layer. Assuming a CE with parallelism configured as *cpf* and *kpf*, *cpf* pairs of inputs and DNN parameters are passed from the on-chip buffers and handled by the process element (assuming in PE_1, which is a sub-module in CE) for multiply-accumulate (MAC) operations, and the generated results are produced along the first output channel dimension. Meanwhile, $PE_2 \sim PE_{kpf}$ are working on the same inputs but different DNN parameters to generate results for the next $kpf-1$ output channels. In this case, *kpf* PEs are instantiated providing a total parallelism factor as $cpf \times kpf$.

Figure 3.1 (b) illustrates how a complete Conv layer is processed by the proposed CE. In this example, the Conv layer has 48 input channels (along the Z-axis of the green cube) and 128 output channels (along the Z-axis of the orange cube). The input and output feature map sizes are both 27×27 (corresponding to the X- and Y-axis), and each convolutional kernel is 5×5 (X-, Y-axis of the blue cube). Assume the CE's parallelism configuration is $cpf = 24$ and $kpf = 16$. In each iteration, there are 24 input channels (the part filled with green color) that are passed to the CE and distributed to 16 PEs for parallel computing. Eventually, the CE generates 16 outputs (the

Algorithm 1: Configurable HLS IP for DNN layer

```
1  for C_i → InCh, C_i+ = C_ii (InCh partitioning) do
2      for C_o → OutCh, C_o+ = C_oo (OutCh partitioning) do
3          for i → KernelHeight , i ++ do
4              for j → KernelWidth , j ++ do
5                  for i → KernelHeight , i ++ do
6                      #pragma HLS dataflow                ▷ Create ping-pong buffer
7                      InFM, weight ← Load_Data( )
8                      for h → OutHeight , h ++ do
9                          for w → OutWidth , w ++ do
10                             #pragma HLS pipeline        ▷ The following loops are unrolled
11                             for coo → C_oo , coo ++ do
12                                 for SelBuf → 1, 2 do
13                                     for cii → C_ii , cii ++ do
14                                         Out[SelBuf][C_o + coo][h][w]
15                                         += weight[SelBuf][coo][cii] ×
                                             InFM[SelBuf][C_i + cii][h + i][w + j]
16                                     end
17                                 end
18                             end
19                         end
20                     end
21                 end
22             end
23         end
24     end
25 end
```

part filled with orange color) as the partial results corresponding to 16 output channels. In this case, we have a total parallelism factor of $24 \times 16 = 384$. By swapping another pair of input feature map and kernel, CE generates the output results corresponding to the next 16 output channels.

Algorithm 1 shows the HLS descriptions of a CE implementation targeting Conv layers. In this design, parallel factors *cpf* and *kpf* are represented by variables C_{ii} and C_{oo}, respectively. In this example, line $11 \sim 18$ describe the key function of the proposed CE as it performs $C_{ii} \times C_{oo}$ multiply-accumulate operations in parallel. When resources allow, we can increase the number of PE instances in each CE IP by increasing Coo to achieve a higher level of parallelism. The achievable performance is proportional to the degree of parallelism because there are no data dependencies following the Coo dimension. Similarly, by increasing C_{ii}, we can also increase the number of input handled by each PE for improved performance. With the configurable HLS IP design, we can instantiate dedicated CEs to handle various layers operations in

Algorithm 2: The REALM algorithm

1 $latency = \alpha \sum_{i=1}^{n} \frac{c_i}{r_i}$

2 $R_{Acc} = \sum_{i=1}^{n} r_i$

3 $\left[\sum_{i=1}^{n} \left(\sqrt{\frac{c_i}{r_i}}\right)^2\right] \left[\sum_{i=1}^{n} (r_i)^2\right] \geq \left[\sum_{i=1}^{n} \sqrt{c_i}\right]^2$

4 $\sum_{i=1}^{n} \frac{c_i}{r_i} \geq \frac{\left[\sum_{i=1}^{n} \sqrt{c_i}\right]^2}{\sum_{i=1}^{n} r_i}$

5 $latency_{min} = \alpha \frac{\left[\sum_{i=1}^{n} \sqrt{c_i}\right]^2}{\sum_{i=1}^{n} r_i}$

6 **REALM:** $\frac{r_i}{r_j} = \frac{\sqrt{c_i}}{\sqrt{c_j}}$

7 Notice: $(a_1^2 + a_2^2 + \cdots + a_n^2)(b_1^2 + b_2^2 + \cdots + b_n^2)$

$\geq (a_1 b_1 + a_2 b_2 + \cdots + a_n b_n)^2$ when:

$\frac{a_1}{b_1} = \frac{a_2}{b_2} = \cdots = \frac{a_n}{b_n}$ satisfies with equality

fulfilling the desired performance within the hardware resource constraints.

3.3.2 Resource allocation

One of the most critical problems in FPGA-based DNN implementation is the unclear resource allocation. Therefore, we propose an algorithm for resource allocation management called REALM to provide allocation guidelines by considering the predefined hardware resource constraints. The main idea of the REALM is shown in Algorithm 2, where we assume the computational demand of the layer i to be c_i and the resource consumed by that layer to be r_i. An increase in r_i results in a proportional increase of the parallel factor that configures the proposed HLS IP (e.g., the CE IP) used in building the layer. With the same logic, we can expect a proportional decrease in the latency of that layer, which can be illustrated in line 1 in Algorithm 2. By considering the resource budget R_{Acc}, formulas shown in line $3 \sim 6$ hold, with α indicating a constant of proportionality.

By taking a close look at these formulas, we have lines 1 and 2 listing the assumptions discussed above. Lines $3 \sim 5$ find a lower bound for the latency

of the overall accelerator under these assumptions. Among them, line 3 is a Cauchy inequality, while line 4 can be generated after simplifying line 3. Line 5 holds when using line 1 and 4. At last, line 6 lists the condition under which minimum latency is obtained, which is hence the resource allocation guideline that we call REALM.

3.3.3 Network compression

Additional optimizations are needed to ensure that our design assumptions are valid and that the LRCN network aligns well with the strengths of the FPGA platform. First, REALM does not consider communication overheads. One of the conditions that can invalidate the assumptions behind REALM is when the computation within the HLS IP itself is not the bottleneck to the latency of a layer, but an external factor like memory is. This can happen in the case of FC layers which have very low CTC ratio (as indicated in Table 3.1). The LSTM layers may also be affected by a similar issue. Also, FPGAs perform favorably with fixed-point operations but not as well as the floating-point operations.

To address these concerns, we adopt network compression technologies to reduce the memory demands of the FC and LSTM layers. We first prune the original LRCN network and reduce the number of output channels in the last two FC layers from 4096 to 256. Also, the two LSTM layers with 1000 hidden units are converted to one LSTM layer with 256 hidden units. We then quantize the weights, biases, and intermediate results into fixed point presentations. These technologies can provide improved DSP utilization and reduce the memory pressure due to smaller memory footprints.

To maintain the inference accuracy, we re-train the compressed LRCN network using Caffe [15] following two steps. First, we train the pruned LRCN network with a floating-point data type. We get a pruned model with comparable accuracy compared to the original LRCN network. We then fine-tune this model and adopt a dynamic quantization scheme as shown in Table 3.2. This quantization scheme allows variable bit-width allocated for the fractional part so that our design can be built to reduce the quantization error. In summary, we use 12-bit quantization for weights and 16-bit quantization for intermediate results. The accuracy results of the resultant networks are

Table 3.2: Dynamic quantization scheme for LRCN

Layers	Output data (total bits, fractional bits)	Weight and Bias data (total bits, fractional bits)
Conv1	16, 4	12, 11
Conv2	16, 7	12, 11
Conv3	16, 8	12, 11
Conv4	16, 9	12, 11
Conv5	16, 10	12, 11
FC1∼FC3	16, 11	12, 11
LSTM	16, 11	12, 11

Table 3.3: Accuracy of the LRCN model after Re-training

Network	Accuracy
LRCN - original (AlexNet + 2 LSTM layers)	43.0%
LRCN - pruned (AlexNet + 1 LSTM layer)	41.8%
LRCN - pruned, quantized (AlexNet + 1 LSTM layer) *implemented on FPGA*	42.0%

summarized in Table 3.3. After network compression, the number of required weight has a 7.8× drop (from 86 Million to 11 Million), while the computation complexity of LRCN dropped around 1.5×, from 2.2 to 1.5 billion operations.

3.3.4 Memory management

To improve the memory access efficiency, the data access patterns need to be simple and straightforward, which means data can be fetched following consecutive locations in memory. To ensure this, the multi-dimensional DNN parameters are re-ordered into a linear sequence that follows the order of computation. This process can guarantee that data locality is exploited when the CE instance accesses weight data and thus improve the data access throughput.

In our design, DNN parameters are quantized and represented by a 12-bit

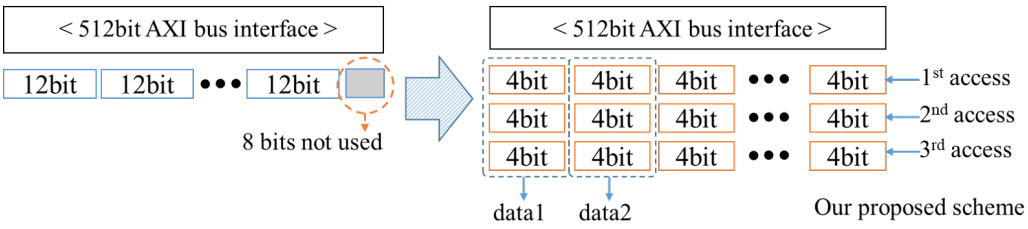

Figure 3.2: Memory bus organization for minimizing bandwidth wastage

fixed-point format, but the memory bus-width is 512, which is not divisible by 12. Using a 12-bit data format means that eight bits in each access need to be discarded. To prevent this, we collect bits from three bus accesses because the number of bits received during the three accesses can be evenly divided into parameters of width 12. The scheme is shown in Figure 3.2. Instead of viewing a single bus access as containing 12-bit parameters, we view it as containing 4-bit fields. Over the course of three bus accesses, we can collect all the required data to form 12-bit parameters without any wastage.

To further hide the effects of off-chip memory access latency, we overlap the computation with communication. When a CE instance is consuming parameters already fetched, fetch requests for the new parameters are sent out concurrently. As shown in Figure 3.3, we design a hierarchical memory system to meet this requirement by instantiating FIFO buffers in the path connecting the on-chip buffers to external memory. The FIFO buffers pre-fetch a few parameters first, and for every buffer row of parameters consumed by CE, a new row is fetched at the back of the buffer. The depth of the FIFO buffer can be adjusted to adapt to different memory access requirements. Besides, the weight buffers are instantiated as ping-pong buffers to hide a few cycles of access time between the proposed CE and the FIFO buffer.

3.4 Hardware Implementation and Comparison

To implement the LRCN accelerator, we need to first pre-process the LRCN source code and set up REALM for resource allocation guidelines. Source codes of LRCN are developed in Caffe, and we convert the Caffe code into a regular C code, which can be executed independently without the Caffe environments. After that, we filter the code by removing all non-synthesizable constructs to satisfy the needs of HLS. These non-synthesizable parts include

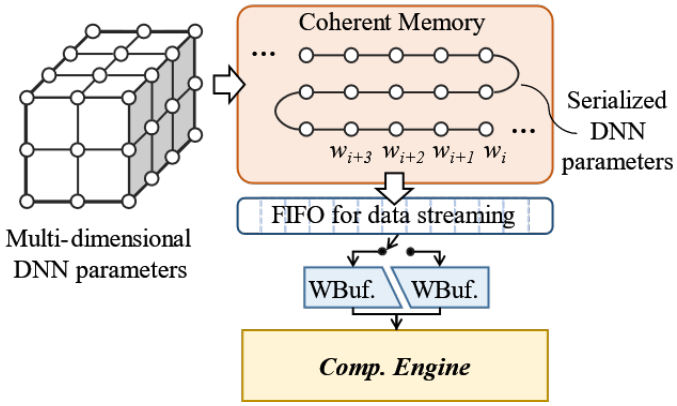

Figure 3.3: Memory hierarchy design for timely data delivery

Table 3.4: FPGA resource consumption

BRAM	DSP	FF	LUT
1508	3130	321165	316250
51%	87%	37%	73%

dynamic memory allocation, recursive functions, and linked list data structures. By processing the source-code, LRCN is ready to go through HLS, and eventually, it can be implemented on FPGA.

During experiments, a Xilinx VC709 board is used for our design following the proposed strategies mentioned in Section 3.3, and Vivado HLS 2016.2 is used for high-level synthesis. To demonstrate the performance of the proposed accelerator, we build a real-time video analysis system that can directly process frames from a commercial webcam and generate sentences for video content descriptions. We use a Tegra TK1 embedded GPU and a Logitech C920 full-HD webcam as the front-end to capture video. We down-sample the captured frames to the size that fits the LRCN network and stream these frames over the internet to the back-end. On the back-end side, the host PC receives the frames and passes them to the proposed LRCN accelerator. Results are sent back to the PC and presented on the monitor. The complete system is shown in Figure 3.4.

After implementation, the resource consumption of the proposed accelerator is shown in Table 3.4. We compare the performance of running the pruned LRCN model on three different hardware platforms, which include

Figure 3.4: The proposed video analysis system includes a front-end (a Tegra TK1 and a webcam) and a back-end (a host PC and a Xilinx VC709 board) for performing real-time video content descriptions.

Table 3.5: LRCN performance comparison

	Frequency	Latency	Speedup	Power	Efficiency
This work	100 MHz	0.040s	4.75×	23.6W	0.94J/image
NVIDIA K80	562 MHz	0.124s	1.53×	133W	16.49J/image
Intel E5-2630	2.6 GHz	0.190s	1.00×	88W	16.72J/image

CPU, GPU, and FPGA. We prepare two versions of the model, the one using floating-point data format for the GPU and CPU and the other using fixed-point data format for the FPGA. In addition, two GPUs in the NVIDIA K80 are used to map the LRCN network (using cuDNN) under the Caffe framework. The CPU version is an optimized implementation (using BLAS) from the Caffe framework.

The performance and power comparisons are provided in Table 3.5. For our FPGA implementation, a power meter is used to measure the consumption of the entire evaluation board during video analysis. For the GPU implementation, power is measured using the command "nvidia-smi", and for the CPU implementation, power is measured by a power meter.

3.5 Conclusion

In this chapter, we presented HLS-based design and optimization strategies

to accelerate LRCN, a DNN for video description and image captioning. These proposed techniques successfully raised the hardware design abstraction level and made the hardware design and optimization process more efficient. The main techniques introduced in this chapter included highly configurable HLS IPs for building neural network layers and a resource allocation strategy called REALM to help map complicated DNNs onto hardware. REALM drives theoretical guidelines for resource allocation given the computational demand of each DNN layer and the available computational resources. We also presented an efficient hierarchical memory system to support data streaming between off-chip and on-chip memory with a flexible memory bus organization design. Results showed that our proposed design achieved 3.1× higher speedup and 17.5× higher efficiency compared to the model implemented on an NVIDIA K80 GPU.

CHAPTER 4

DNNBUILDER: AN AUTOMATED TOOL FOR BUILDING HIGH-PERFORMANCE DNN HARDWARE ACCELERATORS FOR FPGAS

4.1 Introduction

FPGAs have become promising candidates for DNN implementations, but the development of DNN designs on FPGAs still presents significant challenges, such as the tedious RTL programming, the intricate verification problems, and the time-consuming design space exploration process, all of which often hinder FPGAs' adoption by application developers who may have fewer experiences on hardware accelerator design.

The design difficulties also come from the diverse requirements of AI applications and various resource budgets of targeted FPGAs. For example, cloud AI applications require sophisticated resource allocation strategies to accommodate flexible batch-processing and meet throughput requirements with given FPGAs. Edge AI applications usually ask for real-time processing of streaming inputs that limit the FPGAs' ability to batch the data for increased throughput, as the additional latency incurred by the batch process can exceed what is allowed by real-time performance requirements. Also, most edge AI applications require processing high-definition (HD) images/videos, which generates even higher requirements for feature map storage and computation power.

In addition to the HLS-based design and optimization strategies proposed in Chapter 3, we need end-to-end automation tools to keep improving the hardware design efficiency and make hardware accelerators easier to access. In this chapter, we propose DNNBuilder for building DNN designs on FPGAs. It is an automated tool flow that can transform DNN designs from popular deep learning frameworks (such as Caffe [15] and TensorFlow [16]) to highly optimized board-level FPGA implementations with considerations of available computation units, on-chip/off-chip memory, and external memory

access bandwidth in targeted FPGAs. To summarize, the main contributions of DNNBuilder are as follows.

- **An end-to-end automation tool** called DNNBuilder, which provides an integrated design flow from deep learning frameworks to board-level FPGA implementations. With DNNBuilder, users are no longer required to program in RTL or to perform manual resource allocation and optimization for deploying DNNs on FPGAs.
- **A flexible quantization scheme** for smooth tradeoffs between limited resources on FPGAs and desired output accuracy. Our design supports arbitrary quantization for DNN parameters (weights and biases) and activations either within a layer or across layers in DNNs. It also supports binary and ternary networks.
- **A fine-grained layer-based pipeline architecture and a column-based cache scheme** can deliver high throughput (even without batch processing), low startup latency, and low on-chip memory consumption. With the proposed designs, we reduce 7.7× latency and 43× BRAM usage than the conventional structure. These features ensure the millisecond-scale response and HD input support of our design.
- **Highly optimized RTL network components** that can be automatically generated for building DNN layers with high quality. Commonly used loop structures in DNNs are captured by a parameterized PE, which can be configured to deliver the best performance under constraints of given FPGAs.
- **An automatic resource allocation management** that provides computation and memory resource allocation guidelines across network layers. It considers the external memory access bandwidth, data reuse behaviors, computation resource availability, and network complexity.

4.2 The Proposed Automation Flow

DNNBuilder produces board-level FPGA implementations in three steps: *Design*, *Generation*, and *Execution* (Figure 4.1). After networks are determined, RTL codes and corresponding files for running DNN accelerators

on FPGAs can be generated in seconds. In this design, we assume that the targeted network can fit into the targeted hardware device specified by DNNBuilder users.

During the *Design* step, a targeted network is designed and trained using deep learning frameworks, which generally employ CPUs and GPUs. After training, network definition files ("DNN def." in Figure 4.1) and trained parameters are passed to the next step. To ensure design freedom specified by users, the proposed flow supports arbitrary quantization schemes not only for functions within a layer (e.g., Conv, Relu), but also for inputs across layers (e.g., different weight/bias quantizations for layer i and $i+1$) to explore tradeoffs among inference accuracy, resource utilization, performance, etc. One important feature of the *Design* step is that it receives feedbacks from performance estimation ("P. estim.") in *Generation*. If the current DNN design runs slower or consumes more resources than expected, users could update their network designs, such as adjusting quantization schemes or modifying network layers to meet performance requirements. With several iterations between *Design* and *Generation*, the best network configuration can be developed for the targeted FPGA. This feature makes the hardware-software co-design possible.

In the *Generation* step, **network parsing** is the first process for decomposing targeted DNNs from input network models (which include network definitions in *.prototxt* and weight information in *.caffemodel* when using Caffe). Different network layers, e.g., Conv, Pooling, and FC layers, are decomposed and then mapped to our pre-built RTL components. The computational intensive nested loops are captured by parameterized PEs which are introduced in Section 4.3.1. **Automated optimization** works for exploring design space and balancing pipeline stages in DNNBuilder so that our design can achieve maximum throughput performance. We propose an automatic resource allocation scheme (details in Section 4.4) which generates optimization guidelines for parameter adjustment of the pre-built RTL components. Major elements in these guidelines include kernel/channel parallel factors and buffer sizes, and they can be manually modified for expert users. Following the guidelines, **network construction** is responsible for building DNN implementations with the pre-built RTL network components, dataflow controller, and memory instances, which are highly configurable to ensure the adaptability and scalability for various DNNs. **Code generation** generates

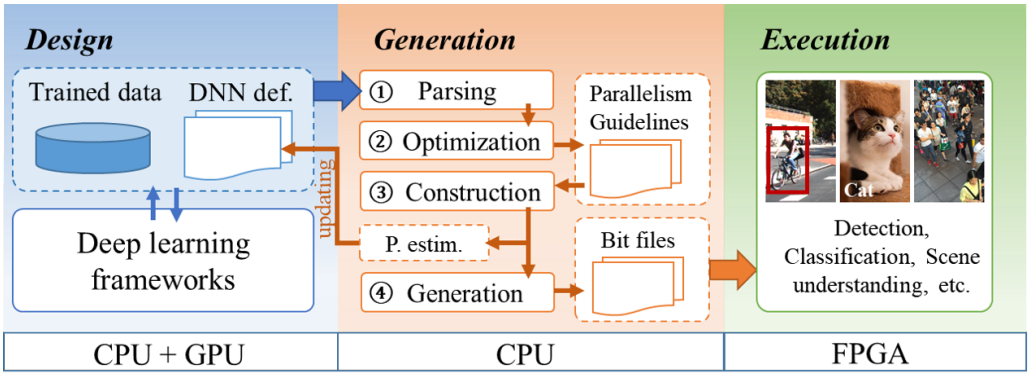

Figure 4.1: Design flow of using DNNBuilder containing DNN design & training (*Design*), network optimization & automated RTL code generation (*Generation*), and FPGA board-level implementation (*Execution*)

accelerator related files for FPGA-based instances.

In the *Execution* step, the DNN accelerator is instantiated in FPGA hardware platforms with unified interfaces, including a FIFO-like data input/output interface and a weight access interface connecting the off-chip memory controller. In this final step, the DNN accelerator is ready for eventual deployment.

4.3 Accelerator Architecture

DNNBuilder generates a pipeline structure where each pipeline stage corresponds to each major neural network layer, like Conv or FC layer, which dominates computation and memory consumption. The rest of the layers, such as batch normalization (BN), scale, and activation layers, are aggregated to the major layers so that we reduce the number of pipeline stages for lower latency.

In Figure 4.2, we present two pipeline stages instantiated on FPGA for computing two Conv layers (i and $i+1$). This design consumes three types of FPGA resources as computation resources (blue area), on-chip memory (green area), and external memory (orange area). Two datapaths are generated for passing input feature maps horizontally and trained DNN parameters vertically to the computation units. To maintain sufficient data supply, we set up two buffers for each pipeline stage as the reshape buffer for keeping slices of the input feature map and the weight buffer for pumping in the

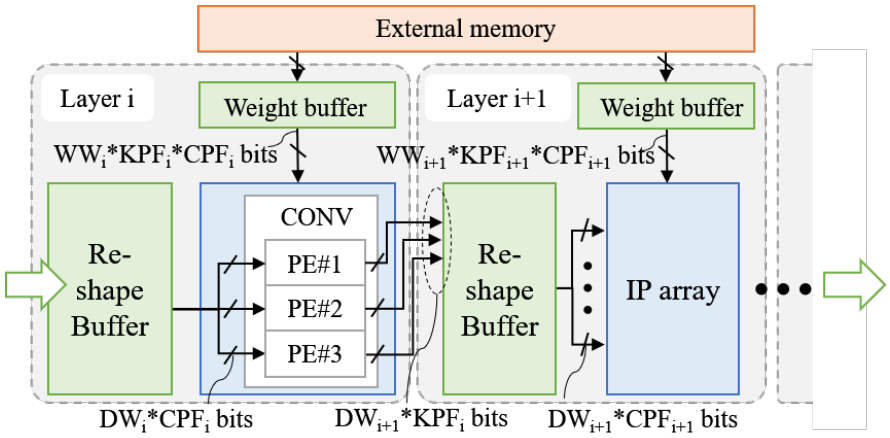

Figure 4.2: Accelerator architecture generated by DNNBuilder

trained parameters from external memory.

We define two parameters, the Channel Parallelism Factor (CPF) and the Kernel Parallelism Factor (KPF). CPF and KPF represent the number of input channels and the number of kernels, respectively, which can be processed in one IP array (a group of RTL network components) inside a pipeline stage. These two factors allow DNNBuilder to implement a two-dimensional parallelism scheme and adjust the resource utilization for each pipeline stage. CPF and KPF are calculated by our resource allocation algorithm, which is discussed in Section 4.4. DNNBuilder also supports flexible quantization schemes. As shown in Figure 4.2, DW_i is the input data bit-width of the i-th layer while WW_i represents the bit-width of weights.

4.3.1 Computation engine design

The core functions in DNNs are carried out by the auto-generated RTL network components (e.g., Conv, FC, Pooling, BN, Relu, etc.), which are the RTL IPs for building the whole network. Since the same for-loop structure is frequently used in Conv and FC, we abstract it as a PE, which can be unfolded in two dimensions corresponding to CPF and KPF. In our design, the CPF and KPF work for unrolling input and output channels, respectively.

Figure 4.3 presents a detailed structure of the PE which is designed for processing CPF number of input feature maps while the number of PEs is decided by KPF. To better explain how the PE works, we take a small-size Conv layer as a case study (notice that CPF and KPF are power of

2 in real case for efficient hardware design). Assuming there is a 4×3×3 input feature map in blue (the left side of Figure 4.3 (a)), it is processed by six 4×2×2 kernels with green color (the middle of Figure 4.3 (a)) with the channel/kernel parallel factors as CPF=2 and KPF=3 (total parallel: 2×3=6). Since CPF=2 and the kernel size is 2, a cube with 2 elements along X-, Y-, and Z-dimension is considered as one tile. Each tile requires four steps of processing following number ① to ④ because only one pixel in the X-Y plane is processed every step. In each step, two pieces of data (along Z-axis) from input feature maps (InFM) are collected (corresponding to the CPF) and they are simultaneously processed by the first three of the six kernels (corresponding to the KPF). In total, six multiply-accumulates are executed in parallel (which equals to CPF×KPF), and the first 3 partial sums (in orange) are generated. These partial sums still need 4 more steps to complete calculation with the next cube in input feature maps along the Z-axis (the second half along the Z-axis surrounded by the dashed line). Figure 4.3 (b) shows the required input data of the PE in one step. Two elements (blue) are fetched from input feature map while six elements (green) are fetched from weights. The reshape buffer and weight buffer provide these data respectively by one memory access. In this example, the order of outputs is illustrated in Figure 4.3 (c), following indexes from 1 to 8. The first three kernels contribute output 1, 2, 5, and 6 while the remainder three kernels generate the output 3, 4, 7, and 8. Figure 4.3 (d) presents the multiply-accumulate operation in the PE with CPF=2 and KPF=3.

Following the idea above, we can design RTL IPs with high-performance and controllable resource overhead. As basic building blocks, these high-quality RTL IPs fundamentally ensure the high-performance design generated by DNNBuilder.

4.3.2 ON-chip/off-chip memory management

In this subsection, we present two techniques to efficiently use the scarce on-chip memory in FPGAs for keeping input feature maps and buffering weight data.

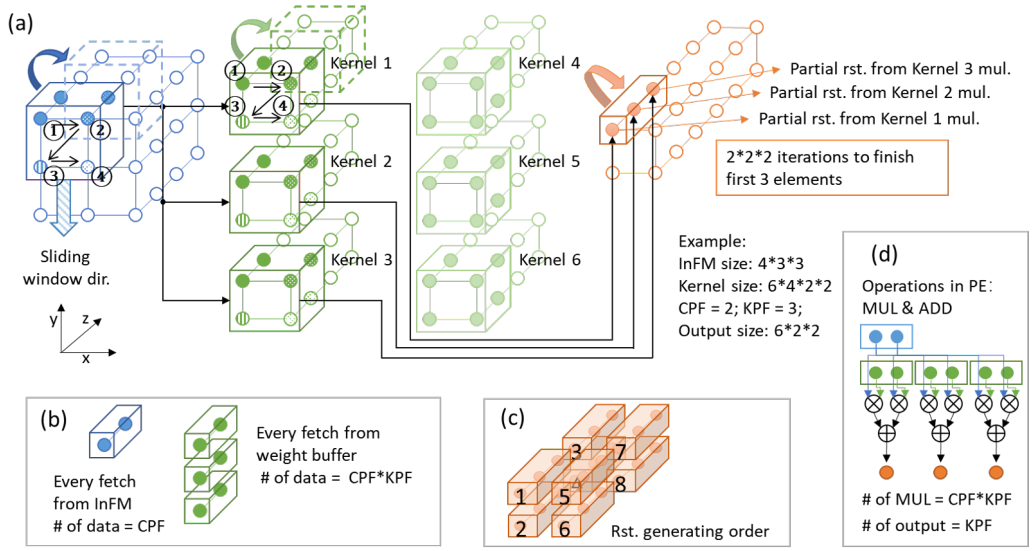

Figure 4.3: The proposed PE in DNNBuilder (aggregated functions are not shown, e.g. BN and ReLU)

1. Column-based cache scheme:

For buffering input feature maps, previous designs (e.g., [87] and [88]) have stored input feature maps on chip to achieve higher throughput and avoid complicated data movements. The size of their input images is relatively small such as 256×256, 32×32, and 28×28 in ImageNet, Cifar100, and MNIST, respectively. But images captured in real-life can easily reach HD, like 1280×720. Although down-sampling may mitigate the issue somewhat, it is not always acceptable, especially for the small object detection. With the HD input, feature maps are enormous and impossible to be stored on chip entirely.

To address this problem, we propose a novel column-based cache scheme for only keeping a subset of the input feature map on chip. Figure 4.4 shows a convolution with kernel size=3 and stride=1. Since slices 1∼3 contribute to the first sliding window operation (from top to bottom), we name the first three slices as column 1. Similarly, column 2 represents the amount of data for the second sliding window operation, so that slices 2∼4 constitute the column 2. DNNBuilder caches at least two columns before starting computing, which allows the kernel to perform the second vertical sliding window operation immediately after finishing the first one. Delay caused by data shortage will not happen by caching one more column. Meanwhile, slice 5 will start

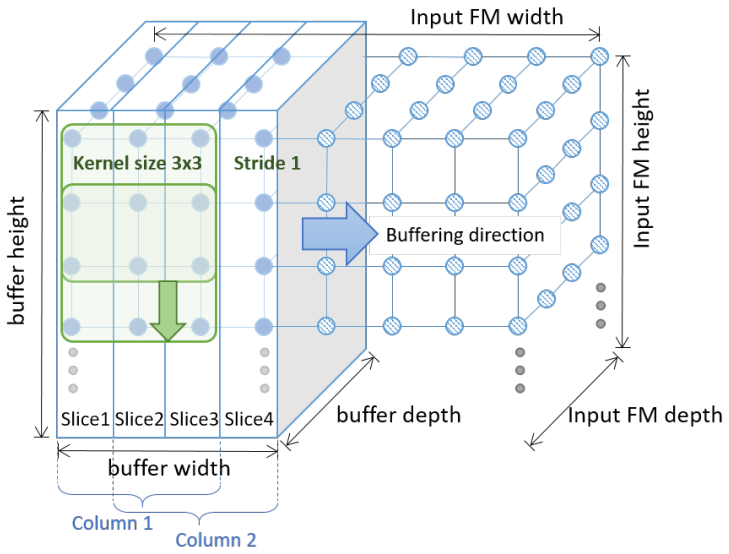

Figure 4.4: The proposed column-based cache scheme

buffering to form the next column (with slices 3∼5) after releasing the room taken by slice 1. In this example, the required size of the reshape buffer (shown in Figure 4.2) equals to the size of two columns (four slices). Since most of the input images have fewer pixels in height than width, we buffer slices in a column way to save on-chip memory. This scheme can be easily extended to row-based processing if needed.

2. Adaptive hierarchical memory system:

To tolerate the delay of off-chip data access, we design an adaptive hierarchical memory system that can insert buffers between the computation-intensive IP array and the external memory (shown in Figure 4.2). The weight buffers are implemented by dual-port RAMs in FPGA for continuously buffering the weights from DRAM. Also, DNNBuilder provides optional ping-pong buffers at the input of each layer. Once the required amount of weights exceeds a certain threshold, these data need to be stored off-chip so that ping-pong buffers are automatically generated to overcome the data shortage problem when fetching data from external memory. Conversely, when users develop a low bit-width quantized DNN, the ping-pong will not be generated, as the required data size is below the threshold.

4.4 Automatic Resource Allocation

One of the most critical problems in FPGA-based DNN implementation is the resource allocation under constraints while seeking optimal performance. To address this problem, we propose an automatic resource allocator for DNNBuilder, which can generate parallel schemes (e.g., CPF and KFP for each layer) and data buffering guidelines (e.g., size of the reshape buffer) with considerations of network complexity, external memory access bandwidth, and data reuse behaviors.

4.4.1 Theoretical guideline

$$L_i = \alpha \frac{C_i}{R_i} \ , \quad \sum_{i=1}^{n} R_i = R_{total} \tag{4.1}$$

$$TP = \frac{1}{max\{L_i\}} \tag{4.2}$$

$$\frac{C_1}{R_1} = \frac{C_2}{R_2} = ... = \frac{C_i}{R_i} \tag{4.3}$$

The theory for maximum throughput performance of the pipeline architecture can be found in Equation 4.1 to 4.3. L_i represents the latency of layer i, while the computation demand (computation complexity) of layer i is C_i and the computation resource consumed by that layer is R_i. Assuming the available resource is R_{total}, the increase of allocated resource for each layer R_i results in a proportional increase in parallelism and eventually lowers the latency for that layer (α is a constant of proportionality related to the hardware working frequency). Since the generated accelerator uses a pipeline architecture, the overall throughput depends on the layer with the maximum computation time (Equation 2). We derive the upper-bound of the throughput from Equation 3, which lists the conditions for achieving the maximum throughput when workloads for all pipeline stages are perfectly balanced. It gives us a theoretical guideline to allocate computation resource for each layer.

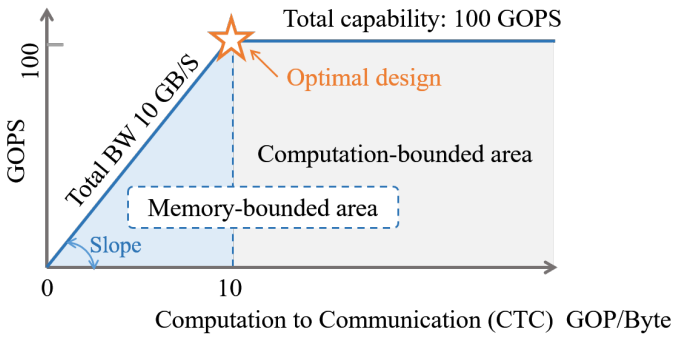

Figure 4.5: Basis of the roofline model

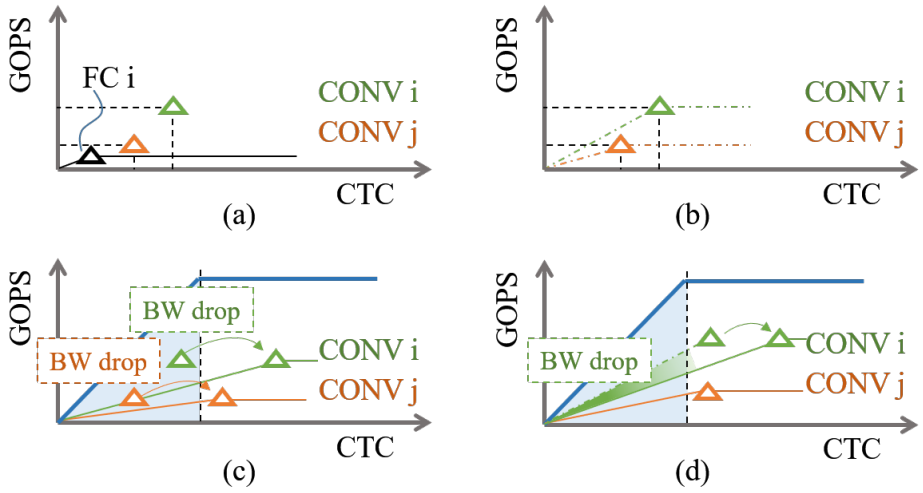

Figure 4.6: Memory bandwidth adjustment

4.4.2 Memory bandwidth adjustment

We use the roofline model adopted by [8] to illustrate the limitation of memory bandwidth. In Figure 4.5, memory access bandwidth is represented by the slope. The area covered by blue is memory-bounded where performance is determined by the memory access latency under given CTC ratio. With a lower CTC ratio, there are fewer data reuse opportunities, meaning more fresh data must be fetched through memory interfaces to maintain GOPS. The right-side gray area is computation-bounded with a larger CTC ratio, where performance is restricted by the available computation resource because memory access bandwidth is not the bottleneck. The optimal design locates at the top-left corner of the roofline, which achieves the maximum throughput (GOPS) with the least bandwidth resource and CTC ratio re-

quired.

Our design intends to adjust the data reuse behavior of each layer so that we change its CTC ratio. For buffering more columns in the proposed column-based cache scheme (Section 4.3.2), we exploit more data reuse opportunities, which means a higher CTC ratio is achieved and it equivalently relaxes the dependency on memory access bandwidth (with a smaller required slope). Eventually, we place each DNN layer close to its optimal design spot and meet the constraints of available bandwidth and on-chip memory.

The procedure of this adjustment is shown in Figure 4.6. During the first step, the layer's computation demand is satisfied by a prorated allocation of the computation resource according to Equation 4.3, and CTC ratio is determined based on the data reuse behavior of each DNN layer. Here we mark three layers in the roofline model in Figure 4.6 (a). Bandwidth resources are first allocated to FC layers to reach their optimal design spots. We do not shrink their required bandwidth, since these layers highly depend on memory bandwidth, and no data reuse can be exploited. If the bandwidth of the targeted FPGA is insufficient for FC layers, we need to use more aggressive quantization (e.g., 8, 4 or even 2 bits) to represent the trained DNN parameters. In this case, DNNBuilder provides feedbacks from *Generation* to *Design* step and updates the network definition, which can be found in Figure 4.1 "updating" arrow. After going through the *Design* step for network retraining, we can avoid a significant accuracy drop.

We then allocate the remainder memory bandwidth to Conv layers as shown in Figure 4.6 (b) (where the FC layer is omitted for conciseness). If Conv layers are memory-bounded as well (Figure 4.6 (c)), it is necessary to cache more columns to get higher CTC ratios so that these layers can be moved to the computation bounded area. In the other case (Figure 4.6 (d)), if the Conv layers are not memory-bounded but the bandwidth is insufficient for them to reach their optimal design spots, we adjust the layer with the highest bandwidth demand (Conv i in our example) and shrink its bandwidth usage by caching more columns of the corresponding feature map even though it locates in the computation-bounded area already. As a result, the Conv i (green marker) is moved to the right with a bandwidth drop (smaller slope). In summary, both Figure 4.6 (c) and (d) introduce bandwidth drop but they focus on two different scenarios.

Algorithm 3: Computation resource allocation

1: Set available computation resource: R_{total} ▷ total DSPs
2: Computation resource for layer i following Equations (4.1)-(4.3):
$R_i = \frac{C_i}{C_{total}} \times R_{total}$ ▷ number of DSPs for layer i
3: Initialize allocated resource for i-th layer (parallelism factor):
4: $R_i = 2^{\lfloor \log_2 R_i \rfloor}$
5: **while** $\sum_{i=1}^{n} R_i \leq R_{total}$
6: Select layer j with maximum $\frac{C_j}{R_j}$
7: **if** $\sum_{i=1}^{n} R_i + 2 \times R_j \leq R_{total}$
8: $R_j = 2 \times R_j$ ▷ double the resource for layer j
9: **else** break
10: **endif**
11: **endwhile**
12: $R_i = CPF_i \times KPF_i$
*CPF, and KPF are power of 2 for efficient hardware implementation

4.4.3 Allocation algorithm in DNNBuilder

To sum up the idea in Section 4.4.1 and 4.4.2, we present a resource allocation algorithm running in DNNBuilder. First, it starts allocating the computation resource shown in Algorithm 3. Since the parallelism factors must be the power of 2, DNNBuilder further fine-tunes the allocation scheme and fills up the gap between actual and the theoretical value (line 5∼line 11). Resource allocated for layer i is represented as R_i, which is also the product of CPF_i and KPF_i. Eventually, DNNBuilder generates parallelism guidelines for building IP instantiations.

In step two, Algorithm 4 allocates the memory bandwidth resource given the constraints of total remainder bandwidth BW_{total}^{conv} for Conv layers (subtracted the bandwidth consumed by FC layers) and the total amount of remainder on-chip memory mem_{total}^{rb} for reshape buffers to keep feature maps (subtracted the memory occupied by weights buffers and reshape buffers in FC layers).

We initialize $Col_i = 1$ (caching one column of the input feature map) and the reshape buffer is implemented by a dual-port RAM with the width of read/write port $width_i^{rd}$, $width_i^{wr}$, and the depth of read port $depth_i^{rd}$ in i-th layer, which can be referred to Figure 4.2. In line 5, Algorithm 4 first satisfies the bandwidth demands of allocated computation resources. PF_i represents the parallel factor, which is equal to R_i. If the required memory bandwidth

Algorithm 4: Memory bandwidth resource allocation

1: Set available memory bandwidth: BW_{total}^{conv}
2: Set available on-chip memory for input feature map: mem_{total}^{rb}
3: Set single DSP's bandwidth usage: BW_R
4: Initialize $Col_i = 1$; size of reshape buffer (e.g. $width_i^{rd}$, $depth_i^{rd}$, and $width_i^{wr}$ according to KPF_i and CPF_i)
5: Allocate bandwidth BW_i for layer i to best satisfy its R_i demand: $BW_i = \frac{PF_i \times BW_R}{H_i^{out} \times Col_i}$
6: **while** $\sum_{i=1}^{n} BW_i \geq BW_{total}^{conv}$ ▷ Conv layer bandwidth overuse
7: Select layer i in Conv layer with maximum BW_i
8: $depth_i^{rd} += \frac{H_i^{in} \times C_i^{in} \times Stride_i}{CPF_i}, depth_{i+1}^{rd} += \frac{H_i^{out} \times C_i^{out}}{CPF_{i+1}}$
9: **if** $\sum_{i=1}^{n} f(width_i^{rd}, depth_i^{rd}, width_i^{wr}) \leq mem_{total}^{rb}$
10: $Col_i = Col_i + 1$ ▷ Cache one more column
11: $BW_i = BW_i \times \frac{Col_i - 1}{Col_i}$
12: **else** ▷ Conv layer bandwidth overuse
13: $depth_i^{rd} -= \frac{H_i^{in} \times C_i^{in} \times Stride_i}{CPF_i}, depth_{i+1}^{rd} -= \frac{H_i^{out} \times C_i^{out}}{CPF_{i+1}}$, break
14: **endif**
15: **endwhile**

exceeds the total available bandwidth, we need bandwidth adjustment (line 6 $\sim$ line 16). It intends to address the problem shown in Figure 4.6 (c) where the layer is in the memory-bounded area and (d) where particular layers excessively consume bandwidth resources. Since the microarchitecture of on-chip memory is significantly different between FPGAs, e.g., RAM18K/36K in Xilinx and M20K in Intel, we use a unified function f shown in line 9 to calculate the number of occupied memory blocks. In this algorithm, H_i^{in}, H_i^{out} and $Stride_i$ represent the height of input and output feature maps and the stride in layer i. C_i^{in} and C_i^{out} represent the number of channels of the input and output feature map in layer i, respectively. Col_i represents the number of cached columns in layer i which relates to the kernel reuse behavior (CTC ratio) and the consumption of on-chip memory mem_i.

4.5 Experimental Results

In this section, we demonstrate the capability and scalability of DNNBuilder by mapping four DNNs onto two FPGAs (XC7Z045 in Xilinx ZC706 and KU115 in AlphaData 8K5) for running edge and cloud applications. We use

Table 4.1: Top-1 Accuracy for image classification

Network	Float32	Fix16	Fix16+f.-t. in *Design*	Fix8	Fix8+f.-t. in *Design*
Alexnet	55.7%	53.3%	55.1% (0.6% ↓)	51.6%	53.4% (2.3% ↓)
ZF	58.0%	56.3%	57.6% (0.4% ↓)	54.2%	56.2% (1.8% ↓)
VGG-16	68.3%	67.0%	69.3% (1.0% ↑)	63.7%	69.2% (0.9% ↑)

Table 4.2: Accuracy for object detection (AP@IOU=0.5)

Network	Precision	Car	Pedestrian	Cyclist	mAP
YOLO (HD)	Float32	88.9%	64.9%	72.5%	75.5%
	Fix16+f.-t. in *Design*	88.9%	65.0%	73.2%	75.7% (0.2% ↑)
	Fix8+f.-t. in *Design*	88.9%	65.2%	72.6%	75.6% (0.1% ↑)

a Yokogawa WT310 digital power meter to measure the power consumption.

4.5.1 DNN models in case study

We build four DNNs using DNNBuilder, which include Alexnet [1], ZF [89], VGG-16 [33] and YOLO [90]. The ZF and VGG-16 are trained on ImageNet dataset [91] with input size 224×224, and Alexnet is trained on ImageNet but uses 227×227 inputs. For VGG-16, to fit in our embedded FPGA (ZC706) and meet the real-time requirement, we have to cut off half of the kernels of Conv layers (except the CONV5) and half of the activations in FC layers. The weights of the FC layers are quantized to 4 bits to reserve memory bandwidth for Conv layers according to the feedback from *Generation* step. For the implementation in KU115 FPGA, we use the original VGG network structure without pruning. Accuracy results are shown in Table 4.1, where column "Float32" means using float32 model without quantization. "Fix16" and "Fix8" show the results of quantized models using 16-bit and 8-bit feature maps and DNN parameters without retraining. "f.-t. in *Design*" represents the accuracy results are collected after retraining and fine-tuning (such as adjusting the quantization following Section 4.4) in the *Design* step. The quantized models may also introduce regularizations, which causes the 1.0% accuracy increase in VGG-16 even compared to the original Float32 version.

Regarding the YOLO network, we modify it from the YOLO-tiny model,

Table 4.3: Evaluation on Xilinx ZC706 (Zynq XC7Z045@200MHz, batch size = 1 for Fix16, batch size = 2 for Fix8)

Network	Complexity (GOP)	Throughput Fix16 (img./s)	GOPS Fix16	Throughput Fix8 (img./s)	GOPS Fix8	DSP Efficiency
Alexnet	1.45	170.0	247	340.0	494	76.3%
ZF	2.34	112.2	263	224.4	526	79.7%
VGG-16 (pruned)	9.45	27.7	262	55.4	524	96.2%
YOLO (HD)	10.6	22.1	234	44.2	468	86.0%

Table 4.4: Resource utilization (Xilinx ZC706)

Network	Utilization			
	LUT (218600)	FF (437200)	BRAM (545)	DSP (900)
Alexnet	86262(39%)	51378(12%)	303(56%)	808(90%)
ZF	87465(40%)	50853(12%)	333(61%)	824(92%)
VGG-16 (pruned)	114521(52%)	61189(14%)	542(99%)	680(76%)
YOLO (HD)	86103(39%)	48853(11%)	333(61%)	680(76%)

which is originally designed for 416×416 input resolution, and adapt it to the KITTI dataset [92] with 1280×384 HD input. Due to the hardware constraint, we change the kernel number of Conv layer 7 and Conv layer 8 from 1024 to 512 to target the real-time detection capability. We choose this model to demonstrate the scalability of DNNBuilder for handling HD inputs. We use 80% of the KITTI provided dataset for training set and the remainder for validation set to perform car, pedestrian, and cyclist detection and show the accuracy results in Table 4.2. We use IOU=0.5 as the threshold to identify true positive cases and mAP to represent the mean average precision of the car, pedestrian, and cyclist categories.

4.5.2 FPGA mapping results

These four DNNs are automatically deployed and optimized following the whole design flow of DNNBuilder. After synthesis by Vivado 2016.4, placement and routing are completed subsequently showing performance and resource utilization in Table 4.3, 4.4, 4.5, and 4.6, respectively. We achieve 200 MHz working frequency in the Zynq XC7Z045 (28nm) and 220-235MHz in

Table 4.5: Evaluation on AlphaData 8K5 (Xilinx KU115) with the batch size pre die = 3, 3, 1 and 4 for Alexnet, ZF, VGG-16, YOLO(HD), respectively. The batch size is doubled for running Fix8 models.

Network	Complexity (GOP)	Freq. (MHz)	Throughput Fix16 (img./s)	GOPS Fix16	Throughput Fix8 (img./s)	GOPS Fix8	DSP Efficiency
Alexnet	1.45	220	1126	1633	2252	3265	76.4%
ZF	2.34	225	759	1776	1518	3552	79.7%
VGG-16	30.94	235	65	2011	130	4022	99.1%
YOLO (HD)	10.6	220	199	2109	398	4218	90.7%

Table 4.6: Resource utilization (Xilinx KU115)

Network	Utilization			
	LUT (663360)	FF (1326720)	BRAM (2160)	DSP (5520)
Alexnet	262360(40%)	177146(13%)	986(46%)	4854(88%)
ZF	268242(40%)	186198(14%)	1162(54%)	4950(90%)
VGG-16	257862(39%)	171616(13%)	1578(81%)	4318(78%)
YOLO (HD)	262356(40%)	165601(13%)	1256(63%)	5286(96%)

KU115 (20nm) without any sophisticated timing adjustment.

The *Batch per Die* in Table 4.5 shows the batch size of Fix16 version in one die of the KU115 FPGA (two dies in total) and we keep the same design in both dies. For the Fix8 version, the performance could be doubled by packing two activations together according to [93]. In summary, DNNBuilder can generate DNN hardware accelerators with performances peaking at 526 GOPS in an embedded FPGA and 4218 GOPS in a mid-range FPGA.

Since DSP is one of the most critical resources of FPGA-based DNN accelerators, we need to carefully evaluate the utilization efficiency of the DSPs. Therefore, we introduce the *DSP efficiency* to exhibit the ratio between actual and theoretical maximum performance of the allocated DSPs. It is defined as:

$$DSP\_efficiency = \frac{Performance}{\beta \times DSP\_num \times freq.} \tag{4.4}$$

Since β multiply-accumulate operations (β=2 in Fix16, β=4 in Fix8) can be handled by one DSP and corresponding logics in one cycle, the denomi-

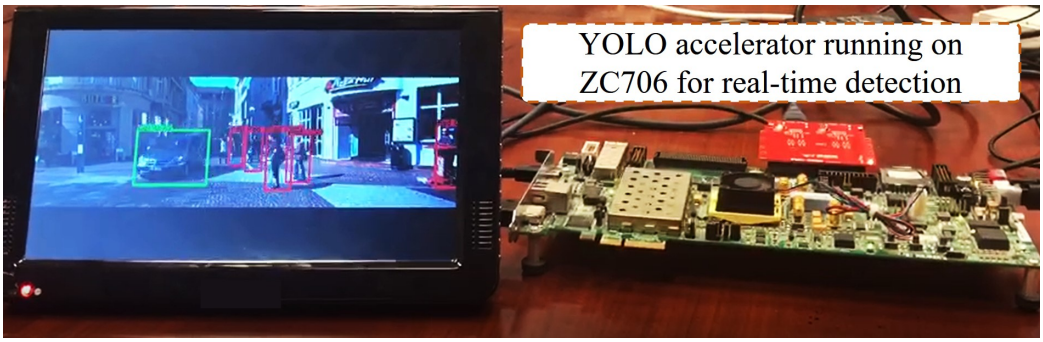

Figure 4.7: Accelerator for real-time car/pedestrian/cyclist detection generated by DNNBuilder

nator equals to the theoretical best performance provided by allocated DSPs under a given frequency. The numerator means the actual achieved performance (GOPS) as shown in Table 4.3 and 4.5. Following Equation (4.4), VGG-16 accelerator achieves the highest DSP efficiency, which is followed by designs for YOLO and ZF, while Alexnet accelerator is in the last place. The reason is that VGG-16 has unified Conv (3×3 with stride 1) and pooling pattern (2×2 with stride 2), which makes Equation (4.3) perfectly satisfied under the constraints that R_i is a power of 2. The more balanced for the latency of each layer, the higher DSP efficiency can be achieved.

4.5.3 Latency and on-chip memory analysis

We take the DNNBuilder generated YOLO accelerator (Fix16) as a case study. In Figure 4.7, a real-time detection for HD video input is running on the targeted FPGA (ZC706). The input video frame with HD resolution (1280×384) is captured by a wide-angle camera and sent to the FPGA at 20 FPS (meaning the frame transmission delay is 50ms). In Table 4.3, this YOLO accelerator achieves a throughput at 22.1 FPS, which is enough for processing the 20 FPS video. Since the proposed fine-grained pipeline architecture and column-based cache scheme are applied, accelerator is launched once the first few columns of input frame are ready (in the reshape buffer). In Figure 4.8, the startup latency is 9.92ms. After 9.92ms, Conv layer 9 keeps generating outputs until 9.04ms after the first frame is fully loaded, so we call this period "output time-slot" which lasts 50+9.04-9.92=49.12ms. This accelerator utilizes 137 BRAMs for keeping columns of feature maps and 333

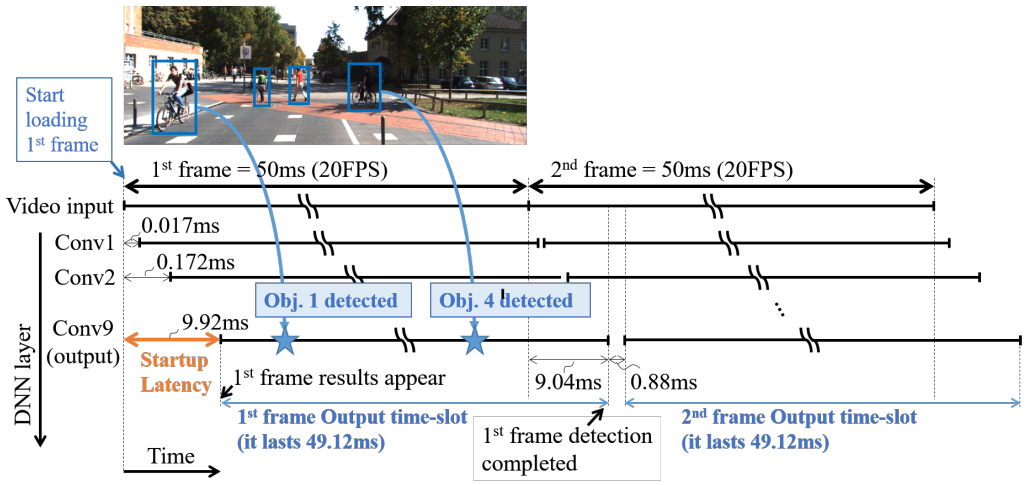

Figure 4.8: Startup latency analysis in pedestrian detection

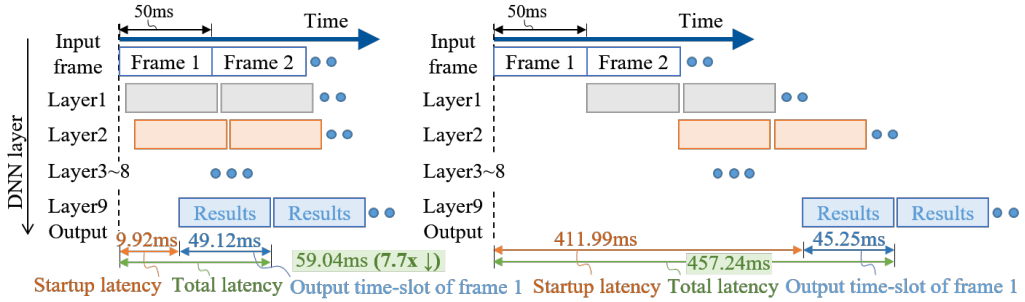

Figure 4.9: Latency comparison between fine-grained (left) and conventional (right) pipeline architecture while running YOLO with HD inputs

BRAMs (Table 4.3) for the whole accelerator. Since four pedestrians are shown in this frame, they are detected one after another from left to right during the"output time-slot".

The advantage of using the proposed fine-grained layer-based pipeline architecture is that we can hide the data transmission delay (generating outputs when the first frame is still loading), and deliver a small startup latency (which is 9.92ms in this case). Detailed comparisons are shown in Figure 4.9. The length of each rectangle represents the latency of one major network layer.

The conventional pipeline architecture (right), with the same 20 FPS overall throughput, waits for the finish of all frame and feature map transmissions at preceding stages so that it suffers a very long latency (457.24ms) for generating all results of the first frame. Conversely, our proposed design starts running layer $i+1$ pipeline stage after we collect several columns of output

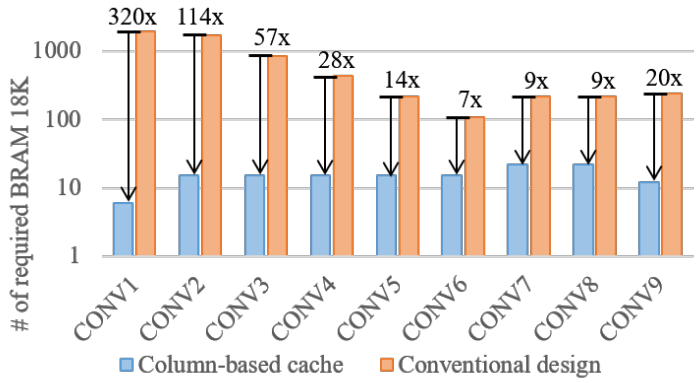

Figure 4.10: On-chip memory demand comparison between column-based cache and conventional design while holding feature maps in YOLO with HD inputs

feature maps of layer i. So, we deliver the latency as 59.04ms, which achieves an amazing 7.7x reduction. The advantage of using column-based cache can be shown in Figure 4.10. We significantly reduce the required BRAM for keeping DNN feature maps. In total, we instantiate only 137 BRAMs in our YOLO accelerator instead of 5920 BRAMs in the conventional pipeline design where feature maps need to be stored completely using on-chip memory. The overall BRAM reduction is $5920 \div 137 = 43$x while the best case happens in the first layer with 320x reduction achieved. The proposed column-based cache ensures the scalability of our accelerator design while using HD or even 4K inputs.

It is the fine-grained layer-based pipeline structure and column-based cache that help us reduce startup latency and BRAM utilization, which are common shortcomings in pipeline structures and hide the video frame transmission time. As a result, DNNBuilder can deliver millisecond-scale responses during object detection and accommodate HD inputs.

4.5.4 Comparison to FPGA & GPU accelerators

We compare our design to the five latest FPGA-based accelerators with classification-oriented DNNs in Table 4.7. For Intel FPGA, the actual DSP utilization in [94] and [95] should be twice as shown in the original paper because one variable-precision DSP block can simultaneously work for two 16-bit multipliers [96]. For the cloud-computing case, the DNNBuilder gen-

Table 4.7: Comparison with existing FPGA-based DNN accelerators

Reference	[94]	[59]	[95]	DNNBuilder
FPGA chip	Arria10-1150	Arria10-1150	Stratix-V + CPU	KU115
FPGA Frequency	303 MHz	385 MHz	200 MHz	235 MHz
Network	Alexnet	VGG	Alexnet	VGG
Precision	Float16	Fix16	Fix16 in FPGA	Fix16 (Fix8)
DSPs (used/total)	2952/3036	2756/3036	512/512 in FPGA	4318/5520
DSP Efficiency	77.3%	84.3%	-	99.1%
Performance (GOPS)	1382	1790	781	2011 (4022)
Power Efficiency (GOPS/W)	30.7	47.8	-	90.2 (180.4)

Table 4.8: Comparison with existing embedded FPGA-based DNN accelerators

Reference	[9]	[66]	DNNBuilder
FPGA chip	Zynq XC7Z045	Zynq XC7Z045	Zynq XC7Z045
Frequency	150 MHz	100 MHz	200MHz
Network	VGG	VGG	VGG
Precision	Fix16	Fix16	Fix16 (Fix8)
DSPs (used/total)	780/900	824/900	680/900
DSP Efficiency	44.0%	69.6%	96.2%
Performance (GOPS)	137	230	262 (524)
Power Efficiency (GOPS/W)	14.2	24.4	36.4 (72.8)

erated design achieves 4022 GOPS using KU115 FPGA. Our design with Fix8 quantization outperforms those in [94], [59], and [95] by 2.91×, 2.25×, and 5.15×, while our Fix16 version outperforms them by 1.46×, 1.12×, and 2.57×, respectively. Although our design uses more DSPs, we deliver the highest DSP efficiency (99.1%) which allows us to slow down the clock frequency for achieving 5.88× higher power efficiency compared to [94].

The design in [95] deploys a DNN accelerator on a Xeon CPU+FPGA system. Layers with low CTC ratio (e.g., FC layers, which are limited by memory access bandwidth on FPGAs) are swapped out to CPU using QPI. Since we can not quantify the equivalent DSP utilization in CPU and the authors fail to mention any power consumption (which should be the sum of CPU+FPGA), we leave the DSP and power efficiency blank for [95].

Major drawbacks of [95] are the large batch size requirement and the resulting high demand for FPGA on-chip memory. It requires large batch size

Table 4.9: Alexnet inference comparison: GPU vs FPGA

Platform	Precision	Batch	Throughput (img./S)	Power (W)	Efficiency (img./S/W)
DNNBuilder (ZC706)	Fix16, Fix8	1, 2	170, 340	7.2	23.6, 47.2
GPU-TX2[6]	Float16	2	250	10.7	23.3
DNNBuilder (KU115)	Fix16, Fix8	3, 6	1126, 2252	22.9	49.2, 98.3
GPU-TitanX	Float32	128	5120	227.0	22.6

to recover the input padding overhead and the low data reuse behavior while running Conv in the frequency domain. Design in [95] may not be feasible for using embedded FPGAs. On the contrary, DNNBuilder can deliver high-performance DNN accelerators on both cloud FPGAs and energy-efficient embedded FPGAs for cloud and edge applications. By evaluating the edge-computing ability, we use the same embedded FPGA in [9] and [66]. Our DNNBuilder generated design reaches the best performance (524 and 262 GOPS in Fix8 and Fix16) and power efficiency (72.8 GOPS/W in Fix8 and 36.4 GOPS/W in Fix16).

We extend our comparison to the latest embedded GPU (TX2) and the high-end GPU (TitanX) in Table 4.9. Because of the real-time requirement of edge applications, we attempt to use the smallest batch size. However, the result of TX2 is using a batch size of two, which is the smallest batch size implementation we could find from NVIDIA's official source. Our design in ZC706 delivers higher efficiency than the TX2-based solution, even without using batch processing. Our design (Fix8) in KU115 delivers 4.35x higher efficiency than the TitanX-based solution (Float32) with a much smaller batch size.

4.6 Conclusion

In this chapter, we presented DNNBuilder, an automation tool for building DNN hardware accelerators on FPGAs with high performance and power efficiency. We proposed a fine-grained layer-based pipeline architecture and a column-based cache scheme for higher throughput, lower pipeline latency, and smaller on-chip memory consumption. We introduced the flexible pro-

cess engine that not only provides optimal implementations of diversified DNN layers but also allows us to adjust the parallelism factors (CPFs and KPFs) to fit in the resource allocation guidelines. We designed an automatic resource allocation algorithm to enable design space exploration and generate parallelism schemes under constraints of computation resource, on-chip memory capacity, and external memory access bandwidth. Because of the above novel designs, we reached the highest throughput performance, peaking at 4218 GOPS (KU115) and 526 GOPS (ZC706), compared to the existing FPGA/embedded FPGA-based solutions. We also achieved higher efficiency (up to 4.35×) than the GPU-based solutions. The novel techniques proposed in this chapter further improve the hardware accelerator design efficiency through automated design process and optimizations for customized DNN accelerators.

CHAPTER 5

SKYNET: EFFICIENT DNN-ACCELERATOR CO-DESIGN STRATEGIES

5.1 Introduction

AI applications require not only high inference accuracy from DNNs, but also aggressive inference speed, throughput, and energy efficiency to meet real-life demands. In addition to building hardware accelerators, applications also rely on hardware-efficient DNN designs when they are deployed onto embedded systems with limited computation and memory resources. As mentioned in Section 2.2, researchers have investigated various network compression and optimization technologies to reduce the redundancy of the DNN structure and make DNN hardware-efficient.

In general, a conventional design process for hardware-efficient DNNs can be summarized in Figure 5.1. It is a top-down design flow that starts from *step 1*: to select a reference DNN by concentrating on accuracy. Such DNNs are typically too complicated for the targeted embedded systems and need to be compressed using network compression technologies in *step 2* and dedicated hardware accelerator design in *step 3*, respectively. Since DNN compression and its hardware implementation are typically carried out in two separate steps, *step 2* and *3* are usually performed in an iterative manner to balance DNN accuracy and hardware performance on targeted devices. Network retraining is also required to regain accuracy after compression before *step 4*. Because of the iterative nature of the process, it is very challenging to cover both inference accuracy in software and deployment efficiency in hardware.

To address the design challenges of building efficient edge AI solutions, we propose DNN-accelerator co-design methods to effectively generate hardware-efficient DNNs from scratch and dedicated FPGA accelerators. Following these methods, we propose SkyNet, a co-design strategy for handling de-

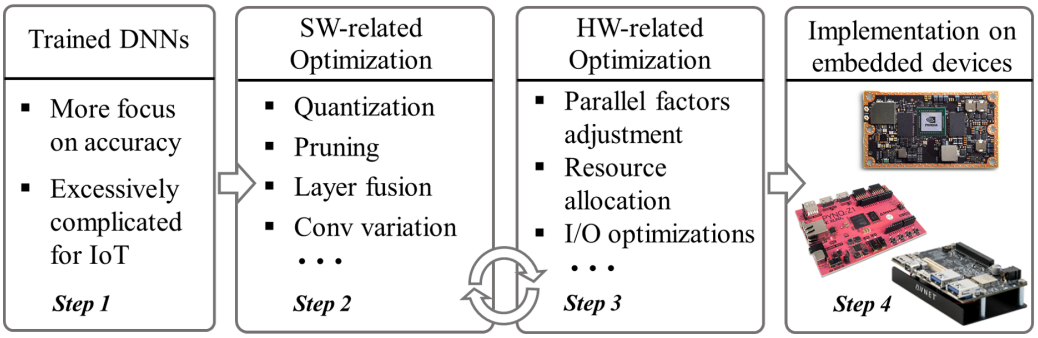

Figure 5.1: A conventional design flow for edge AI solution. It contains DNN compression in *step 2* and hardware accelerator design in *step 3*. Challenges appear between *step 2* and *3* where iterative explorations are necessary to balance DNN accuracy and performance on targeted devices.

manding AI applications, including object detection and tracking. The contributions of this chapter are summarized as follows:

- We propose a simultaneous DNN-accelerator co-design methodology, which includes (1) a bottom-up hardware-efficient DNN design approach and (2) a top-down approach for high-performance FPGA accelerator design. We introduce a unified intermediate representation called Bundle to construct the desired DNN and accelerator for the targeted edge AI application.

- For DNN design, we introduce a three-step bottom-up design approach to construct hardware-efficient DNN with an adequate understanding of the hardware constraints. For accelerator design, we introduce a fine-grained tile-based pipeline architecture to provide dedicated support for the co-designed DNN.

- Following the co-design methods, we propose SkyNet and demonstrate it by winning the DAC-SDC [97]. SkyNet achieved the highest overall score regarding accuracy, throughput, and energy efficiency and won the first place winner awards for both GPU and FPGA tracks.

5.2 DNN-Accelerator Co-Design

The proposed co-design flow is intended to solve two design problems simultaneously: the bottom-up DNN model exploration and the top-down DNN

accelerator generation. For DNN models, we start from basic hardware-aware building blocks and gradually construct DNNs to reach desired inference accuracy; for DNN accelerators, we follow a fixed architecture and optimize configurable parameters to pursue the most efficient DNN implementation.

5.2.1 Co-design space

There is a large design space for DNN design, such as the number and types of DNN layers, layer-wise configurations, layer interconnection styles, etc. Similarly, the design space for hardware accelerators is also enormous, including IP instance categories, IP reuse strategies, quantization schemes, parallel factors, data transfer behaviors, buffer sizes, etc. Therefore, to cover both the DNN model and accelerator design, the co-design space is exponentially greater than any of the above, which requires effective techniques to find high-quality solutions. In this chapter, we propose a co-design space to summarize the configurable parameters for DNN-accelerator co-design.

The variables in the proposed co-design space are summarized in Table 5.1. For FPGA accelerator, we use IP-based design strategy mentioned in Chapter 3 [18] and Chapter 4 [10]. Each IP supports a basic DNN layer type (e.g. Conv, Pooling), which must be instantiated and configured if the DNN model contains such type of layer. L is the total number of DNN layers. IP_1 to IP_m represent the available configurable IP templates. $p_j (1 \leq j \leq n)$ represents the configured IP instance, where the configurable parameters include parallelism factor PF_j and quantization scheme Q_j. $< l_j^1, \cdots l_j^z >$ represent the layers for which an IP instance p_j is used in accelerator to conduct the computation. The vector $< f_{ch_1}, f_{ch_2}, \cdots, f_{ch_L} >$ represents the expansions of channel depth through the entire DNN. In addition, ds_1 to ds_k represent down sampling layers with a down sampling factor f_{ds_i}. The combination of these parameters can specify the DNN model and the accelerator design.

5.2.2 A unified intermediate representation

One of the most critical challenges is the lack of unified intermediate representations between the DNN and its accelerator design. So, the DNN and

Table 5.1: Key Variables in the co-design space

Variables	Explanation	Effect
L	Total number of layers	A, P, R
$IP_1, IP_2, \cdots, IP_m$	IP templates for DNN building	A, P, R
$p_1, p_2, \cdots, p_n$	Labels for IP instances	P, R
$\langle PF_j, Q_j \rangle$	Configuration for $p_j (1 \leq j \leq n)$	A, P, R
$\langle l_j^1, \cdots, l_j^z \rangle$	The layers where p_j is used	A, P
$< f_{ch_1}, f_{ch_2}, \cdots, f_{ch_L} >$	Channel expansion factors	A, P, R
$ds_1, ds_2, \cdots, ds_k$	Down-sampling layers	A, P, R
f_{ds_i}	Down-sampling factor	A, P, R
A: Accuracy, P: Performance, R: Resource		

accelerator designs are still in two separate directions using respective metrics. For example, a particular DNN tries to achieve higher inference accuracy by adding extra layers and channel numbers. It may help achieve higher inference accuracy, but this design causes extra calculations and reduces the expected hardware performance.

To help balance the different metrics between the DNN and the accelerator design, we propose a unified intermediate representation called Bundle. From a software perspective, a Bundle is a set of sequential DNN layers, which can be repeatedly stacked for constructing DNNs. From the hardware perspective, a Bundle is a set of IPs to be implemented on hardware. As shown in Figure 5.2, a Bundle contains three DNN layers cascaded from top to bottom. DNN models are built by replicating, shaping, and configuring this type of Bundle in a bottom-up manner. Figure 5.2 shows three replications of the same Bundle, and each of them may vary in input/output data dimensions. Between Bundles, we reserve down-sampling spots for feature map (FM) size compression.

When implemented on hardware, a Bundle also represents a combination of the IP instances for handling DNN layer computations. The IPs within one Bundle are organized based on our proposed fine-grained tile-based pipeline architecture (which will be introduced in Section 5.5), to deliver optimized low-latency designs.

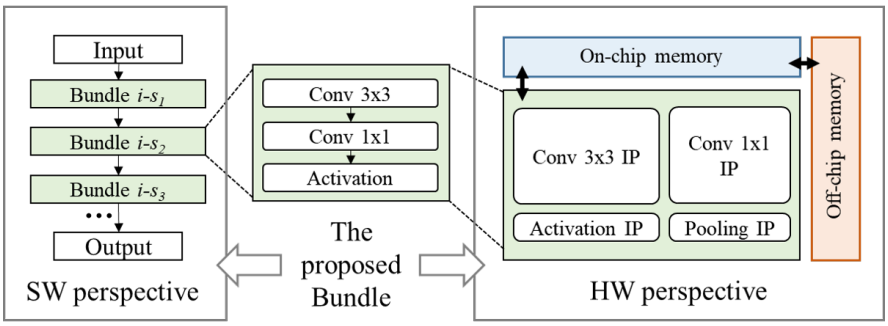

Figure 5.2: The proposed unified intermediate representation called Bundle.

5.3 A Bottom-up DNN Design strategy

We propose a bottom-up approach to leverage the hardware-efficient DNN design for embedded systems. It is a three-stage approach as shown in Figure 5.3.

5.3.1 Stage 1 Bundle selection and evaluation

This flow starts with building Bundles, the hardware-aware basic blocks for DNN construction. To capture the hardware constraints, Bundles need to be evaluated on targeted embedded systems for collecting realistic latency and resource utilization results. In the first stage, we enumerate DNN components (such as Conv, pooling, activation layers, etc.) and assemble them into Bundle $1 \sim n$. Each Bundle is then implemented and evaluated in targeted hardware devices for hardware performance metrics. To get their potential accuracy contributions, we build DNN sketches with fixed front- and back-end structures based on given tasks and respectively insert one type of Bundle (with replications) in the middle. We limit one type of Bundle for one DNN sketch to guarantee its hardware efficiency. Then, DNN sketches are fast trained using the targeted datasets to find out the ones with relatively high accuracy.

By targeting the object detection task, for example, we concatenate an input resizing unit (front-end) and a bounding box regression (back-end) with the selected Bundle to build a DNN sketch. The number of training epochs may vary from different datasets as a 20-epoch-training can distinguish sketches using the DAC-SDC dataset (with 100K images), while five epochs are enough if using the Cifar-10 dataset. We have also seen similar

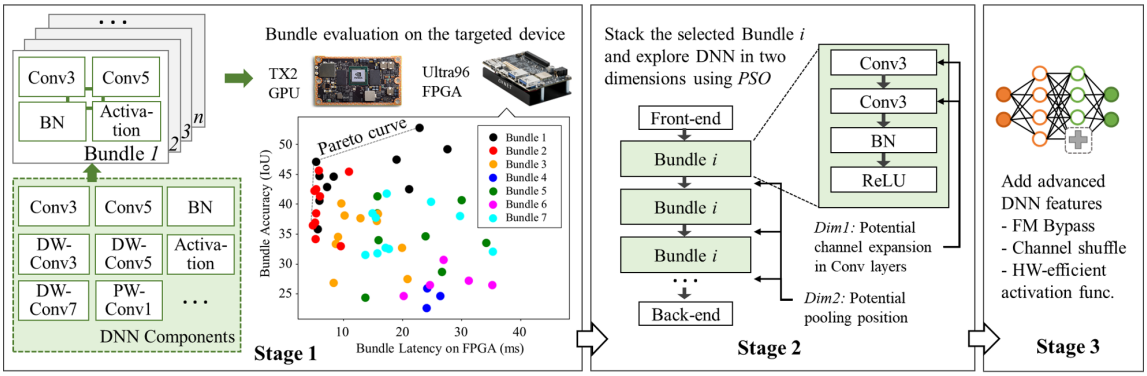

Figure 5.3: The proposed bottom-up DNN design flow to deliver hardware-efficient DNNs for embedded systems in three stages.

strategies in [76] to distinguish candidates by a 25-epoch-training on a subset of the ImageNet dataset. At last, the most promising Bundles located in the Pareto curve are selected for the next stage.

5.3.2 Stage 2 Hardware-aware DNN search

During the DNN search stage, its inputs include the software and hardware metrics (e.g., DNN accuracy and throughput performance) and the targeted hardware platforms. Its outputs are DNN candidates built by the proposed Bundle, which can meet the software and hardware requirements.

To solve such a multi-objective optimization problem, we propose a group-based particle swarm optimization (PSO) evolutionary algorithm to discover proper DNN candidates since literature has demonstrated the validity of using evolutionary methods to discover DNNs with state-of-the-art accuracy [98, 99]. From the design methodology perspective, this solution can be extended to support other optimization algorithms and meet the needs of different scenarios.

In the proposed group-based PSO algorithm, each individual DNN is regarded as a particle, and all active DNNs during the search contribute to the swarm. Since we only use one type of Bundle in each DNN, DNNs composed of the same type of Bundle are considered in a particle group. In order to maintain evolution stability, a DNN only evolves within its own group. We label the group optimal position as P^i_{group} within the i-th group, meaning such DNN has the best fitness value evaluated under given conditions. We

Algorithm 5: The bottom-up DNN design with PSO

$P \leftarrow$ InitialPopulation($\mathcal{M}$, $\mathcal{N}$)
while $itr < \mathcal{I}$ **do**
  FastTraining(P, e_{itr})
  $Fit^i_j \leftarrow$ **GetFitnessVal**(P) //evaluate all candidates
  for *each group* i **do**
    GroupRank(i) //rank candidates in group i
    $N^i_{group} \leftarrow$ **GroupBest**(i) //select the best one in group i
    //get the group best position
    $P^i_{group}(fv1, fv2) \leftarrow$**GetPosition**($N^i_{group}$)
    for *each candidate* $n^i_j(itr)$ *in group* i **do**
      //rank n^i_j across all passing iterations
      LocalRank(i, j)
      $N^{ij}_{local} \leftarrow$ **LocalBest**(i, j)
      //get the local best position
      $P^{ij}_{local}(fv1, fv2) \leftarrow$**GetPosition** ($N^{ij}_{local}$)
      //get the current position
      $P^i_j(fv1, fv2) \leftarrow$**GetPosition** ($n^i_j(itr)$)
      //get the velocity toward the local and the group best
      $V_{local} \leftarrow$**GetV**(P^i_j, P^{ij}_{local})
      $V_{group} \leftarrow$**GetV**(P^i_j, P^i_{group})
      $n^i_j(itr+1) \leftarrow$ **Evolve**($n^i_j(itr)$, V_{local}, V_{group})
    end
  end
end

denote a DNN particle j within group i as n^i_j and each n^i_j has a pair of feature vectors $(fv1, fv2)$ to illustrate two hyper-parameters regarding the DNN structure. $fv1$ represents the number of channels of each Bundle replication; $fv2$ describes the pooling position between Bundles. Both feature vectors with dimensions equal to the number of stacked Bundles in n^i_j, and both of them affect accuracy and hardware performance. To locate the best DNN candidates, we propose Algorithm 5 with the following major components.

Population generation:

An initial network population P (a set of DNN candidates) is generated with $\mathcal{M}$ groups and $\mathcal{N}$ networks for each group. The search contains $\mathcal{I}$ iterations and in the itr-th iteration, all networks are fast trained for e_{itr} epochs, where e_{itr} increases with itr.

Latency estimation:

We perform a platform-specific latency estimation. For GPUs, we directly measure the inference latency on the training GPU and scale latency to the targeted GPU for deployment if the target GPU differs from the training one. For FPGAs, we follow a predefined IP-based DNN accelerator template [61] for hardware performance evaluation. Layer-specific IPs are implemented in hardware and shared by corresponding DNN layers. To maximize the performance, IPs are configured to fully consume the available resources. We then collect the end-to-end performance and resource overhead of each DNN from an FPGA high-level synthesis tool.

Fitness value:

After network training and latency estimation, we calculate the fitness value for each network n_j^i as:

$$Fit_j^i = Acc_j^i + \alpha \cdot (Est(n_j^i) - Tar) \tag{5.1}$$

where Acc_j^i is the validation accuracy of n_j^i and $Est(n_j^i)$ represents the latency on hardware; Tar is the targeted latency. Parameters α ($\alpha < 0$) is used to balance between network accuracy and hardware performance.

Velocity calculation and particle update

In standard PSO, the updated velocity of a particle is calculated every iteration based on the current velocity and the velocities toward the local and the global best positions. Particles can move to a better position with assigned probabilities following the updated velocity. Similarly, in our case, DNNs in the same group update their positions (meaning network structures represented by feature vectors) based on the current design, the local best design (the best one across all passing iterations), and the group best design. To determine the velocity toward the local best V_{local} and the group best V_{group}, we compute the differences between positions of current and the local/group best designs. Since each position is represented by $(fv1, fv2)$, position differences can be captured by the mismatch of layer expansion factors $fv1$ and pooling spots $fv2$, respectively. Then, with the velocities known, we start

evolving the current network by updating its position toward the local and the group best by a random percentage.

5.3.3 Stage 3 Feature Addition

More advanced DNN design features are added if hardware metrics allow. For example, we can include a bypass from low-level features to high-level features along with FM reordering [90] to improve small object detection. We can also replace ReLU with ReLU6 [100] to enhance hardware efficiency. More discussions are provided in the next section.

5.4 The SkyNet

Following the proposed flow, the best Bundle is selected as a combination of 3×3 depth-wise Conv layer (DW-Conv3 [48]), 1×1 point-wise Conv layer (PW-Conv1), batch normalization layer (BN [101]), and ReLU6. By repeatedly stacking this Bundle, we generate three backbones shown in Table 5.2 to handle the object detection challenge in DAC-SDC. These networks share the same chain structure but with different configurations of FM bypass. For model A, no bypass is included; for the model B and C, output FMs of Bundle #3 are fed into the Bundle #6. SkyNet also adapts the YOLO detector head [102] by removing the classification output and uses two anchors for bounding box regression.

By examining the DAC-SDC training data, we keep a record of the size ratio between the output bounding box and the input image and present a distribution diagram in Figure 5.4. It clearly shows that 91% of the objects to be detected are less than 9% of the original input image size, and 31% of them are even smaller than 1% of the input image size. It means the majority of objects inside this dataset are small objects. So, we add FM bypass and reordering to enhance the ability to detect small objects (in model B and C). The bypass helps to keep small object features in the later part (closer to the output layer) of the DNN by adding low-level high-resolution FMs. Also, it is beneficial to have multiple FMs (from different layers) before generating the bounding boxes. Since the bypass crosses a pooling layer (highlighted in Figure 5.5), we use reordering (shown in Figure 5.6) to align the size of

Table 5.2: The SkyNet architecture with number of channels shown in the bracket. Each convolutional layer except the last one is followed by a BN and a ReLU (omitted for conciseness).

<table>
<tr><td colspan="4">Configurations of SkyNet</td></tr>
<tr><td>A</td><td colspan="2">B</td><td>C</td><td>Bundle</td></tr>
<tr><td colspan="4">input (3×160×360 color image)</td></tr>
<tr><td colspan="3">DW-Conv3 (3)
PW-Conv1 (48)</td><td>#1</td></tr>
<tr><td colspan="4">2×2 max-pooling</td></tr>
<tr><td colspan="3">DW-Conv3 (48)
PW-Conv1 (96)</td><td>#2</td></tr>
<tr><td colspan="4">2×2 max-pooling</td></tr>
<tr><td colspan="3">DW-Conv3 (96)
PW-Conv1 (192)
[Bypass Start] FM Reordering (768)</td><td>#3</td></tr>
<tr><td colspan="4">2×2 max-pooling</td></tr>
<tr><td colspan="3">DW-Conv3 (192)
PW-Conv1 (384)</td><td>#4</td></tr>
<tr><td colspan="3">DW-Conv3 (384)
PW-Conv1 (512)</td><td>#5</td></tr>
<tr><td></td><td>[Bypass End]
FM Concatenated
DW-Conv3
(512+768)
PW-Conv1 (48)</td><td>[Bypass End]
FM Concatenated
DW-Conv3
(512+768)
PW-Conv1 (96)</td><td>#6</td></tr>
<tr><td colspan="4">PW-Conv1 (10)
Back-end for bounding box regression</td></tr>
</table>

original FM (generated by the Bundle #5) and the low-level feature without losing information.

The other feature to improve hardware efficiency is the ReLU6, which clips output range to $[0, 6]$. Since ReLU6 generates a much smaller data range compared to the original ReLU ($[0, +\infty)$), fewer bits are required to represent intermediate FMs. It also helps better implement lower-precision floating point in embedded GPUs and fixed-point data format in embedded FPGAs.

5.5 A Top-down Accelerator Design Strategy

The top-down accelerator follows a well-defined hardware architecture template called *Tile-Arch*, which is a fine-grained tile-based pipeline accelerator

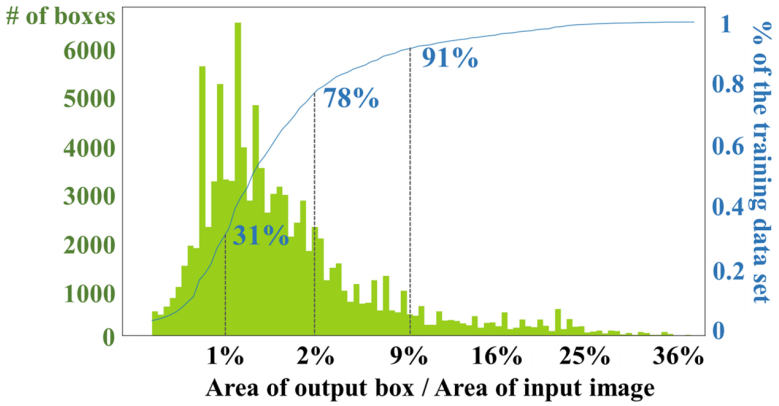

Figure 5.4: The distribution of bounding box relative size in DAC-SDC training dataset. We capture the bounding box relative size by computing the ratio of output bounding box size divided by the input image size. The green bars show the ratio distribution, and the blue curve shows the corresponding cumulative distribution.

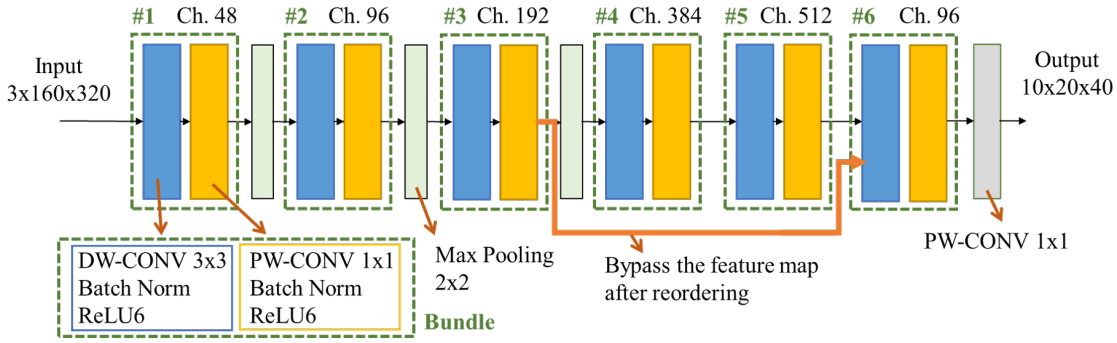

Figure 5.5: SkyNet backbone (model C in Table 5.2) generated by stacking six of the selected Bundle (circled by green dashed line) with DNN components as: DW-Conv3, PW-Conv1, BN, and ReLU6. The number of output channels is listed on top of each Bundle denoted as Ch. Three 2×2 pooling layers are inserted. The bypass is highlighted in orange, which passes FMs generated by the Bundle #3 directly to the last Bundle. The FM reordering is also performed along with the bypass.

architecture template. It can deliver low latency designs and exploit maximum resource-saving. This template has the following features:

- Layer-level IP reuse: we adopt a folded overall structure, where the DNN layers are computed sequentially on FPGA by reusing IP instances across layers. It can maximally exploit resource reuse, which is especially crucial for embedded FPGAs with tight resource budgets.

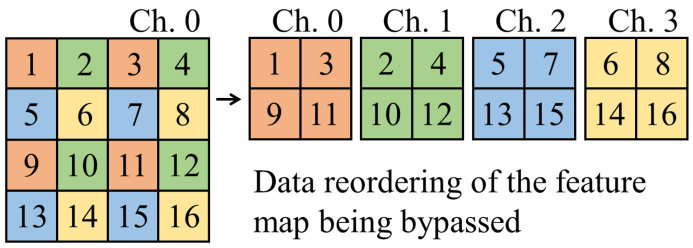

Figure 5.6: Feature map reordering from $1 \times 4 \times 4$ to $4 \times 2 \times 2$ with shrunken width and height but expanded number of channels. There is no information loss compared to pooling operation. In addition, this reorder pattern also ensures larger receptive field.

- Tile-level IP reuse: resulting from layer-level IP reuse, the intermediate data between layers are partitioned into tiles of common size across all layers, and an IP instance is reused for multiple tiles. It allows direct data transfer between IP instances of subsequent layers without on-/off-chip memory access.
- Tile-level pipelining: since data tiles within a layer do not have data dependencies, we can leverage tile-level IP pipelining within and across consecutive layers.

Figure 5.7 (a) shows an example of the top-level diagram of the proposed template architecture. In this example, the Bundle contains IP instances including Conv 3×3, 1×1 and Pooling. On-chip data buffers are allocated in BRAM for intra-Bundle communication, while off-chip data buffers are allocated in DRAM for inter-Bundle communication. Figure 5.7 (b) illustrates the tile-level pipelining for computation in one Bundle with four tiles. Following the top-down approach, parameters of the proposed architecture can be configured to adapt to different FPGA devices and to maximize the performance of FPGA accelerators.

5.6 Experimental Results on DAC-SDC

DAC-SDC features a single object detection challenge for embedded systems, which include embedded GPUs (NVIDIA TX2) and FPGAs (Pynq-Z1 and Ultra96) with very low energy consumption. The goal is to consider the

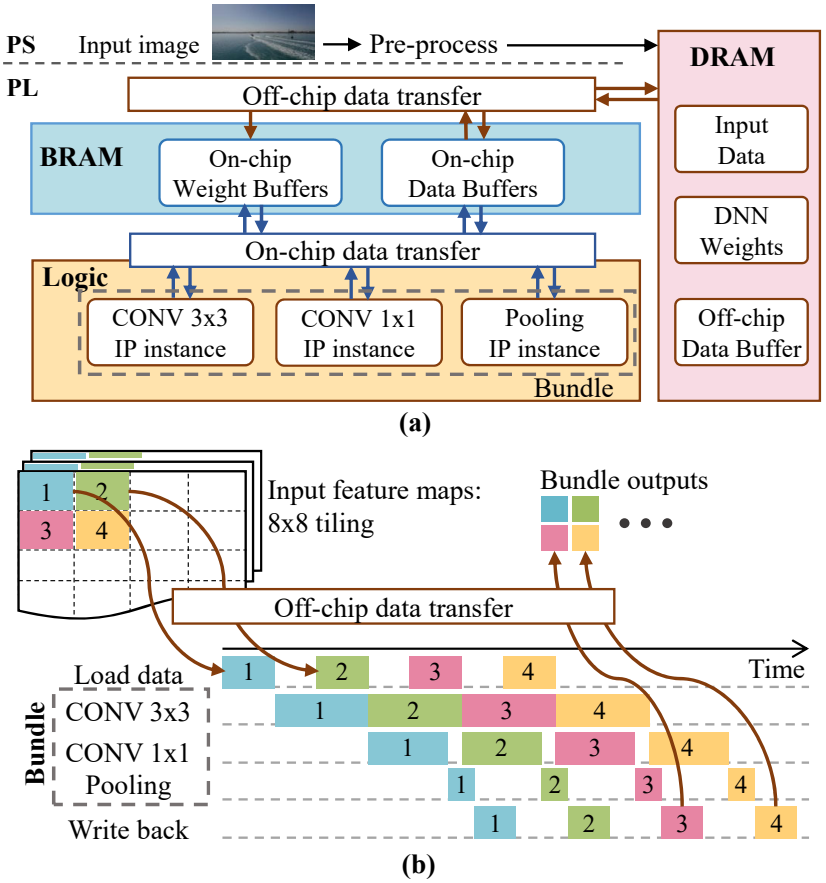

Figure 5.7: *Tile-Arc*: a low latency FPGA accelerator template with (a) a top-level diagram of the proposed architecture and (b) an example of tile-based pipeline structure.

most appropriate needs of UAV applications, such as the capability of real-time processing, energy efficiency, and detection accuracy. To better reflect real-life challenges, images of the dataset are captured by UAVs in the real environment. The whole dataset is divided into two parts: the training dataset with 100,000 images with objects of interest across 12 main categories and 95 subcategories, and the hidden test set for official evaluation with 50,000 images that only the contest organizers could access. In DAC-SDC'19, 52 GPU and 58 FPGA teams participated worldwide, creating very intense competition. Our SkyNet design has successfully delivered the best inference accuracy and total score for both GPU and FPGA tracks.

Table 5.3: Validation accuracy of SkyNet.

DNN Model	Parameter Size	IoU
SkyNet A - ReLU	1.27 MB	0.653
SkyNet A - ReLU6	1.27 MB	0.673
SkyNet B - ReLU	1.57 MB	0.685
SkyNet B - ReLU6	1.57 MB	0.703
SkyNet C - ReLU	1.82 MB	0.713
SkyNet C - ReLU6	1.82 MB	**0.741**

5.6.1 Ablation study

We perform an ablation study on the DAC-SDC dataset to analyze these three configurations of SkyNet (Model A, B, and C listed in Table 5.2). By combining two activation functions (ReLU and ReLU6), six configurations of SkyNet are evaluated. We train these models in an end-to-end fashion using multi-scale training with the learning rate starting from 1e-4 to 1e-7. We apply stochastic gradient descent (SGD) to update parameters. To further enrich the training data, we use data augmentations to distort, jitter, crop, and resize inputs with size 160×320. The accuracy results are presented in Table 5.3, where SkyNet C - ReLU6 reaches the highest IoU (0.741) on the validation set. Therefore, we use this model as the proposed design for the following experiments.

5.6.2 Evaluation Criteria

Comprehensive evaluations are introduced in DAC-SDC, covering detection accuracy (IoU), throughput (FPS), and energy consumption. To identify the best design, a total score is calculated following Equation 5.2 to 5.5. Assuming there are I registered teams and K images in the test set, the IoU score for team i, denoted as R_{IoU_i}, is calculated as:

$$R_{IoU_i} = \frac{\sum_{k=1}^{K} IoU_{i,k}}{K} \tag{5.2}$$

For energy, $\bar{E}_I$ is denoted as the average energy consumption of all I entries when performing DNN inference on the test dataset (Equation 5.3). The

energy score of team i (ES_i) is then computed using Equation 5.4 relating to the ratio between average energy and the energy consumed by this team. x is set to 2 and 10 for FPGA track and GPU track, respectively. Eventually, the total score, denoted as TS_i, is calculated in Equation 5.5 including both inference accuracy (R_{IoU_i}) and energy consumption (ES_i).

$$\bar{E}_I = \frac{\sum_{i=1}^{I} E_i}{I} \tag{5.3}$$

$$ES_i = max\{0, 1 + 0.2 \times log_x \frac{\bar{E}_I}{E_i}\} \tag{5.4}$$

$$TS_i = R_{IoU_i} \times (1 + ES_i) \tag{5.5}$$

5.6.3 GPU Implementation

For the TX2 GPU implementation, we keep all network parameters using Float32 to maintain the best inference accuracy. Since most of the compute-intensive parts of DNN inference are handled by NVIDIA cuDNN, which leaves little space for customized improvement, we start optimizing our design on a system level.

The whole procedure of running SkyNet contains four steps: 1) input fetching from the flash storage in a unit of the batch; 2) image pre-processing, which includes input resizing and normalization; 3) DNN inference; and 4) post-process to generate bounding boxes and buffer results in DDR memory. The most straightforward way is to execute these steps in serial but with the cost of low resource utilization and poor throughput performance. In our design, we first merge step 1 and 2 in pre-processing and enable multithreading technology to execute these steps in a pipelined fashion shown in Figure 5.8.

We use NVIDIA System Profiler (L4T) to capture the latency results. On average, the proposed system-level optimizations enable a 3.35× speedup compared to the original serial design and help our design reach the highest throughput performance, peaking at 67.33 FPS.

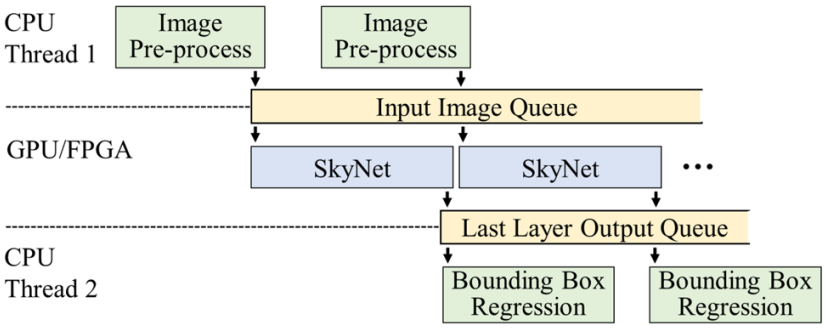

Figure 5.8: Task partitioning in SkyNet implementation on TX2 GPU and Ultra96 FPGA.

Table 5.4: Validation accuracy results regarding different quantization schemes during FPGA implementation

Scheme	Feature Map	Weight	Accuracy (IoU)
0	Float32	Float32	0.741
1	9 bits	11 bits	0.727
2	9 bits	10 bits	0.714
3	8 bits	11 bits	0.690
4	8 bits	10 bits	0.680

5.6.4 FPGA Implementation

To implement DNNs on FPGA, we suffer even scarcer resource budgets, as the theoretical peak performance provided by Ultra96 FPGA (144 GOPS @200MHz) is much lower than the TX2 GPU (665 GFLOPS @1300MHz). By using the proposed bottom-up design flow, hardware limitations have already been captured by the Bundle design, and the Bundle is instantiated on FPGA as a single customized hardware IP. Since the proposed network is structured by the same type of Bundle, this IP can be shared across different layers to cope with the resource constraints. Still, we need more optimizations to further enhance the performance.

Quantization, batch process, and tiling:

Since fixed-point representation is more favorable in FPGA design, we quantize the FMs and weights from Float32 to fixed point and explore different quantization schemes in Table 5.4. After quantization, the SkyNet backbone suffers different levels of accuracy drop from 1.4% to 6.1% in scheme 1 to 4.

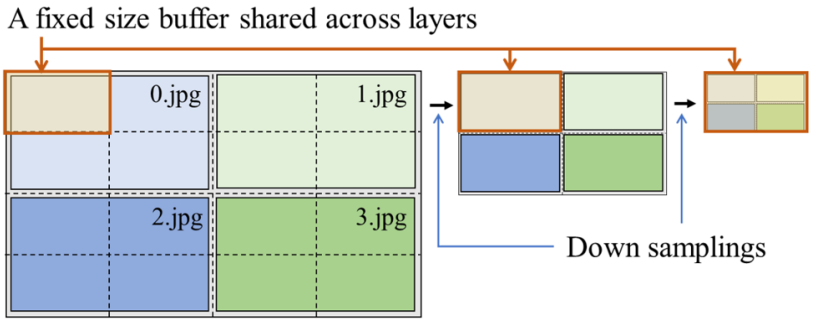

Figure 5.9: The proposed batch and tiling design to increase the data reuse opportunity and avoid on-chip memory waste.

We finally pick scheme 1, as accuracy has a higher weight in the total score calculation (Equation 5.5).

Since network parameters can not be accommodated by the FPGA on-chip memory (which is BRAM with only 0.95 MB available), we have to store them in the external memory (DRAM). Still, it makes the memory access bandwidth a bottleneck. To mitigate the bandwidth demand, the input batch process is applied to exploit data reuse opportunities, where a certain number (which equals to the batch size) of input images are assembled before sending to FPGA for DNN inference. So, task size (the number of images being processed at one time) increases while consuming the same amount of network parameters from DRAM.

With a larger batch size, the process of network inference asks for a larger amount of FPGA on-chip memory to buffer intermediate FMs. Since our implementation is based on an IP-shared structure, buffers instantiated on FPGA are shared by different layers, which means the buffer may not be large enough for the FMs generated by the first few layers while it may be too large for the last few layers as FMs get smaller after pooling.

To solve this problem, we propose an input tiling and batch scheme as shown in Figure 5.9. Four inputs are stitched to form a larger input that can be processed as an entirety. With the tiling and batch process, it is possible to use one shared buffer across different layers without changing its size. The proposed solution inherits the benefit of the batch process to allow better reuse of DNN weights, and it eliminates the possible waste of unused buffer space.

Layer fusion, memory hierarchy, and task partitioning:

To avoid dealing with the floating-point operations (e.g., inverse-square root) in the BN layer, we use layer fusion to merge both parameters from Conv and its successive BN offline. So, there are no separated BN layers nor expensive floating-point operations required during DNN inference.

With hardware resources shared by DNN layers, the intermediate results need to be swapped in/out between on-chip and external memory. To boost the performance, we instantiate the selected Bundle on hardware and implement a five-stage pipeline with Load, EXE-CONV3, EXE-CONV1, EXE-Pooling, and WriteBack stages. By using ping-pong buffers between memory and computation units, data transfer (in the Load and WriteBack stages) can be fully overlapped by computation latency. Regarding the data transfer between adjacent execution stages (with the "EXE" prefix), we keep data on-chip without going through external memory.

To fully utilize the available computational resource, we also implement task partitioning on the Ultra96. The whole design is shown in Figure 5.8, which is highly similar to our GPU design. Workloads are distributed to both CPU and FPGA and creating a system-level pipeline. With all three tasks (pre-processing, SkyNet inference, and post-processing) overlapped, our FPGA design can reach 25.05 FPS.

5.6.5 Result Comparison

After implementing the proposed DNN on GPU and FPGA following the proposed co-design strategies, our designs are evaluated by the DAC-SDC organizers using the hidden test set. As shown in Table 5.5 and 5.6, we present the comparison results with the top-3 teams in DAC-SDC'19 and '18. In our GPU design, SkyNet outperforms all other competitors by delivering the best accuracy (0.731), throughput performance (67.33), and total score (1.504). In terms of the FPGA design, SkyNet also reaches the best accuracy and gets the highest total score.

Table 5.5: GPU final results from DAC-SDC'19 and '18 using the hidden test set with 50K images, evaluated by a TX2 GPU.

Team Name	**IoU**	**FPS**	**Power(W)**	**Total Score**
Results from 2019				
SkyNet (ours)	**0.731**	**67.33**	13.50	**1.504**
Thinker [103]	0.713	28.79	**8.55**	1.442
DeepZS [104]	0.723	26.37	15.12	1.422
Results from 2018				
ICT-CAS [105]	0.698	24.55	12.58	1.373
DeepZ [106]	0.691	25.30	13.27	1.359
SDU-legend [107]	0.685	23.64	10.31	1.358

Table 5.6: FPGA final results in DAC-SDC'19 and '18 using the hidden test set with 50K images. Designs in 2019 are evaluated on a Ultra96 FPGA while designs in 2018 use a Pynq-Z1 FPGA.

Team Name	**IoU**	**FPS**	**Power (W)**	**Total Score**
Results in 2019				
SkyNet (ours)	**0.716**	25.05	7.26	**1.526**
XJTU_Tripler [108]	0.615	50.91	9.25	1.394
SystemsETHZ [109]	0.553	**55.13**	6.69	1.318
Results in 2018				
TGIIF [110]	0.624	11.96	4.20	1.267
SystemsETHZ [111]	0.492	25.97	**2.45**	1.179
iSmart2 [112]	0.573	7.35	2.59	1.164

5.7 SkyNet Extension on GOT-10K Object Tracking

Since SkyNet can deliver real-time object detection on embedded systems, we set up experiments on the GOT-10k benchmark [113] to demonstrate its potential for object tracking. GOT-10k is a large and highly diverse database for generic object tracking with rich motion trajectory and comprehensive coverage of object classes. Models are evaluated with two metrics in GOT-10k as average overlap (AO) and success rate (SR). AO is defined as the mean of IoU between prediction and ground truth bounding boxes, while SR

Table 5.7: Performance of SiamRPN++ trackers on GOT-10k with different backbones evaluated on single NVIDIA 1080Ti.

Backbone	AO	$SR_{0.50}$	$SR_{0.75}$	FPS
AlexNet	0.354	0.385	0.101	52.36
ResNet-50	0.365	0.411	0.115	25.90
SkyNet	0.364	0.391	0.116	41.22

is defined as the proportion of predictions where the IoU is beyond some threshold. During an evaluation, Got-10K only provides the ground truth bounding box in the first frame and expects trackers to keep tracking the same object for subsequent frames by predicting bounding boxes. The predictions will then be evaluated by the Got-10K server. In this section, we integrate the SkyNet backbone with two state-of-the-art trackers (SiamRPN++ and SiamMask) and evaluate its capability for real-time tracking.

5.7.1 Evaluation Using SiamRPN++

Siamese network is one of the most popular network structures for building object trackers. The Siamese trackers locate the object by the correlation between features extracted from the exemplar image and search image, where DNN-based feature extraction plays an important role. SiamRPN++ [114] is the first Siamese tracker that has been proven to profit from using DNN backbones with different capacities as long as they are properly trained. To evaluate the performance of different backbones, we train three SiamRPN++ trackers with AlexNet, ResNet-50, and SkyNet backbones on GOT-10k. We maintain the size of exemplar and search images as 127×127 and 255×255 (128×128 and 256×256 for SkyNet for better implementation efficiency), respectively, and we set the learning rates to start from 1e-3 to 1e-5. Results are shown in Table 5.7 where SkyNet achieves nearly the same quality (AO and SR) as the ResNet-50 backbone but much better speed (1.59× faster).

5.7.2 Evaluation Using SiamMask

SiamMask [115] is another Siamese tracker which outperforms SiamRPN++ by incorporating image segmentation for object tracking tasks. Since the seg-

Table 5.8: Performance of SiamMask trackers on GOT-10k with different backbones evaluated on single NVIDIA 1080Ti.

Backbone	AO	$SR_{0.50}$	$SR_{0.75}$	FPS
ResNet-50	0.380	0.439	0.153	17.44
SkyNet	0.390	0.442	0.158	30.15

mentation information is not provided, it cannot be directly trained with the GOT-10k dataset. Instead, we perform training using Youtube-VOS dataset [116] and apply object tracking on Got-10K to compare the performance of different backbones using the SiamMask structure. We maintain the same input size setup as Section 5.7.1 and apply the learning rates from 1e-3 to 1e-4. As shown in Table 5.8, the proposed SkyNet backbone outperforms ResNet-50 in all metrics when using SiamMask tracker with better tracking quality and 1.73× speedup.

5.8 Conclusion

In this chapter, we proposed a DNN-accelerator co-design method for delivering efficient edge AI applications. The proposed co-design method includes a bottom-up hardware-efficient DNN design approach to capture hardware limitations using realistic hardware feedback and a top-down approach for high-performance FPGA accelerator design. We then introduced a unified intermediate representation called Bundle to effectively balance the hardware and software metrics. By applying the proposed co-design method, we proposed SkyNet for running object detection on embedded systems. SkyNet was demonstrated by the low power object detection challenge on the IEEE/ACM DAC-SDC and won the first place winner award for both GPU and FPGA tracks. We also extended SkyNet to handle object tracking tasks, and it delivered 1.60× and 1.73× higher FPS, and 37.20× smaller parameter size with comparable accuracy when compared to the state-of-the-art Siamese trackers with ResNet-50 backbone. By enabling DNN-accelerator co-design, the solutions introduced in this chapter can solve more challenging co-design problems by delivering highly optimized DNNs and accelerators simultaneously.

CHAPTER 6

F-CAD: A CUSTOMIZED ACCELERATOR DESIGN FLOW FOR EMERGING EDGE VR APPLICATIONS

6.1 Introduction

VR is a major area in which AI technologies play essential roles, and it urgently needs efficient hardware acceleration. The emerging VR applications are compute- and memory-intensive and require real-time and high-quality image rendering on edge devices (e.g., VR headsets). Take codec avatar as an example. It is one of the most impressive breakthroughs that enables immersed communications in VR with photo-realistic and three-dimensional human appearances and real-time expressions [29]. This emerging application helps achieve VR telepresence with more effective communications with not only speaking and listening but also facial expressions and body languages.

The latest DNN model to render codec avatar (introduced in Section 2.1.3) contains multiple DNN branches and complicated layer dependencies with more than 13.6 GFLOP and 7.2 million parameters. It is required to serve with real-time response and high throughput (90 or even 120 FPS) for smooth user interactions. We present the whole codec avatar system in Figure 6.1, where all the information (e.g., a wry smile and a furrowed brow) of the transmitter (TX) will be encoded, transmitted, and decoded after reaching the receiver (RX) to generate the TX's codec avatar for high-fidelity social presence. Among them, the decoder is the most complex module, occupying 90% of the calculations required by the entire system. Without effective optimizations, it can easily become the bottleneck and hinder the smooth running of VR telepresence.

However, deploying codec avatar decoders on VR headsets presents significant challenges. It is compute- and memory-intensive. It also requires a higher refresh rate (90 or even 120 FPS) compared to non-VR applications (30 FPS) to prevent motion sickness and real-time response for smooth user

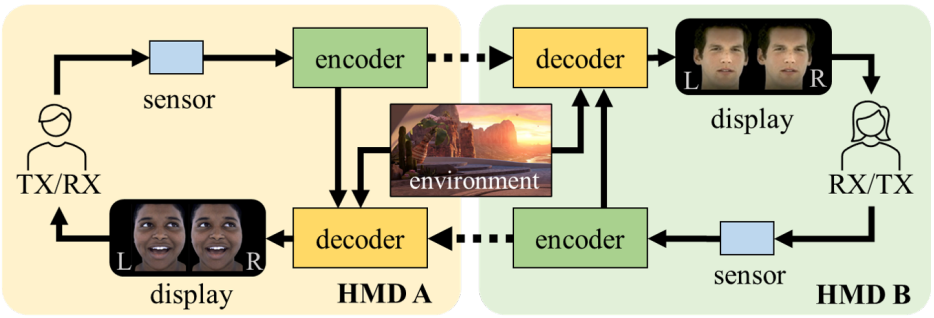

Figure 6.1: Social interactions with codec avatars using head-mount devices (HMDs) in VR. The TX's information is captured by the built-in sensor, compressed into a TX code by the encoder, and passed through the network. The RX then starts decoding based on the virtual environment and the state of the RX, and eventually displays TX's information on the HMD [20].

interactions. In this case, the strategy of using a large batch size becomes infeasible, as the extra delay in collecting batch inputs may fail to meet the real-time requirement. In addition, emerging codec avatar decoders start adopting complicated multi-branch DNNs with customized neural network layers to generate different components of the codec avatar, such as one branch for facial geometry and another for textures. These branches may have very different requirements. Such unique challenges make codec avatar decoders difficult to be handled effectively by existing hardware accelerators. Evaluated by a state-of-the-art commercial SoC processor (Snapdragon 865 [117]) and two recently published DNN accelerators from academia (DNNBuilder [10] and HybridDNN [72]), we have found that all these accelerators failed to deliver satisfactory performance and efficiency required by this compelling VR application.

So, we propose F-CAD, a new automation tool for building customized hardware accelerators to accelerate emerging VR applications at the edge. We focus on codec avatar decoding in this chapter as an important and practical use case of F-CAD to deliver the optimized hardware accelerators by meeting specific performance targets under resource budgets. To summarize, the main contributions are as follows.

- This is the first work that focuses on building an automated design flow with rapid hardware accelerator design and exploration to leverage VR avatar applications for resource-constrained devices.

- We propose a novel elastic architecture. It is a new accelerator template with two-dimensional expansion capability to flexibly support multi-branch DNNs with complicated layer dependencies and a well-constructed architecture unit to support up to three-dimensional parallelism (3D parallelism) for high throughput and efficiency.
- We define a multi-branch dynamic design space to cover all possible hardware design combinations. It allows F-CAD to obtain the optimized solution with the best achievable performance by considering customized requirements and resource constraints.
- We integrate a DSE engine to leverage efficient explorations within the predefined space and deliver the best accelerator by considering various customized constraints, such as available resources, maximum parallelism, maximum batch size, different branch priorities, etc.

6.2 Design Challenges of accelerating VR codec avatar

6.2.1 Background of codec avatar

Codec avatar is formulated as a view-dependent Variational Auto-Encoder (VAE) framework [29, 30]. As described in Figure 6.2, the encoder E takes sensor-captured images X as inputs and generates an l-dimensional latent (TX) code z:

$$z \leftarrow E(X), z \in \mathbb{R}^l \tag{6.1}$$

where E is a trained DNN transforming spatial features into a latent code. This code and the view code (v, indicating the RX's view direction) are processed by the decoder D to generate graphic components of avatars. In Figure 6.2, D outputs facial geometry and view-specific texture information:

$$M, T \leftarrow D(z, v), M \in \mathbb{R}^{n\times 3}, T \in \mathbb{R}^{w\times h} \tag{6.2}$$

where D is a multi-branch DNN with deconvolution-like structures for reconstructing TX's realistic VR appearances. M represents the facial shape comprising n-vertices, and T is the view-dependent RGB texture with $w \times h$

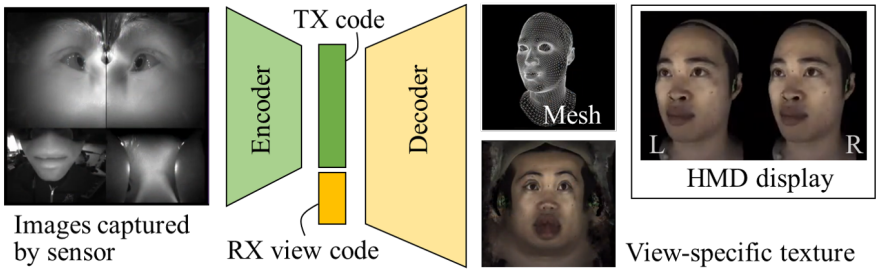

Figure 6.2: The VAE framework adopted by the Codec Avatar application.

resolution. In state-of-the-art designs, decoders are much more complicated than encoders, which contribute more than 90% of operations of the whole VAE framework. Therefore, decoders urgently need high-performance and efficient accelerators.

6.2.2 The targeted decoder structure

In this chapter, we target a state-of-the-art codec avatar decoder for facial animation. It is shown in Table 6.1 with three branches (Br.) for generating facial geometry (3D vertices), UV texture (a 2D surface of a 3D model following U- and V-axis), and warp field (specular effects), of which the second and third branches have a common front part. The input of Br. 1 is reshaped from a 256-dimensional latent code, while the input of Br. 2 and 3 is the combination of both latent and view code to have texture appearance conditioned by different view angles. We adopt **C**, **A**, and **U** to represent the customized Conv, activation, and up-sampling layer, respectively, and we use $\times$ to indicate the number of repetitions. In total, the decoder contains 13.6 GFLOP and 7.2 million parameters without repeatedly counting the shared part. Unlike general DNNs, the decoder introduces complex data dependencies by adopting multi-branch structures and HD intermediate results for high-quality VR avatar textures. Other features come from the customized Conv, where each output pixel has its dedicated bias (as known as untied bias) instead of sharing one bias across pixels within the same output channel.

Table 6.1: Network architecture of the targeted decoder

Br.	**[Input size]→Network→[Output size]**		**GFLOP**	**Parameters**
1	[4,8,8]→[**CAU**]×5+**C**→[3,256,256]		1.9 (10.5%)	1.1M (12.1%)
2	[7,8,8]→	[**CAU**]×2+**C**→[3,1k,1k]	11.3 (62.4%)	6.1M (67.0%)
3	[**CAU**]×5+	**C**→[2,256,256]	4.9 (27.1%)	1.9M (20.9%)

6.2.3 Accelerator design challenges

The unique multi-branch feature and customized layers from codec avatar decoders cause complicated dataflows and high compute and memory demands during inference, which make existing DNN accelerators ineffective. Challenges include the enormous and unevenly distributed computations and the substantial memory footprints (with the intermediate feature map size up to $16 \times 1024 \times 1024$). This becomes even more challenging for hardware accelerators with limited resources but aiming at real-time response with high throughput performance.

We select three existing accelerators from industry (Snapdragon 865 SoC [117]) and academia (DNNBuilder [10], HybridDNN [72]) to accelerate codec avatar decoding. For 865 SoC, we run the targeted decoder shown in Table 6.1. Since DNNBuilder and HybridDNN have not supported the customized Conv, we create a mimic decoder by replacing the customized Conv with the conventional one while keeping the rest of the network structure unchanged. The mimic decoder has a highly similar structure but 3.7% less computations, which can still provide insights to identify bottlenecks of the existing accelerator designs. During the evaluation, we use two performance indicators, including FPS (to indicate the throughput) and efficiency (the ratio between actual and theoretical peak throughput as Equation 6.3 to evaluate whether an accelerator works efficiently). β represents the number of operations handled by one multiplier in one clock cycle. For example, $\beta = 2$ for 16-bit operands in FPGA where multipliers are implemented by DSPs.

$$EFFI = \frac{GOP\ per\ Second}{\beta \times \#\ of\ Multiplier \times FREQ} \tag{6.3}$$

As shown in Table 6.2, the 865 SoC only delivers 35.8 FPS even though it integrates an AI engine for DNN workloads. The major bottleneck is its

Table 6.2: Evaluations by 865 SoC (@1450MHz) [117], DNNBuilder (@200MHz) [10] and HybridDNN (@200MHz) [72] when running the targeted decoder

	Scheme	**Utilization**	**FPS**	**Efficiency**
865 (8-bit)	-	-	35.8	16.9%
DNNBuilder (8-bit)	1	DSP: 644, BRAM: 723	30.5	81.6%
	2	DSP: 1044, BRAM: 861	30.5	50.4%
	3	DSP: 1820, BRAM: 1197	30.5	28.8%
HybridDNN (16-bit)	1	DSP: 512, BRAM: 576	12.1	77.5%
	2 & 3	DSP: 1024, BRAM: 1120	22.0	70.4%

limited cache size, which causes frequent data transfers and severely restricts performance. So, its overall efficiency barely reaches 16.9%. We then target three FPGAs (Xilinx Z7045, ZU17EG, and ZU9CG corresponding to Scheme 1, 2, and 3) with increasing resource budgets and let DNNBuilder and HybridDNN generate accelerators with unfolded and folded accelerator structures for running the mimic decoder, respectively. DNNBuilder achieves a slightly lower throughput (30.5 FPS) and a much higher efficiency (81.6%) using the Z7045 FPGA in Scheme 1 compared to the 865 SoC. Its unfolded structure allows dedicated DNN layer acceleration for higher design specificity. Still, it fails to scale using more resources in Scheme 2 and 3 and suffers deteriorating efficiency. For HybridDNN, we adopt a 16-bit mimic decoder as 8-bit models are not supported. The scalability is slightly better than DNNBuilder when handling more abundant resources in Scheme 2. However, its folded structure and coarse-grained configuration prevent further scaling in Scheme 3, and it generates an accelerator with the same size as that from Scheme 2.

6.3 The Proposed F-CAD Design Flow

We propose F-CAD to address these challenges and deliver customized hardware accelerators for codec avatar decoding. As shown in Figure 6.3, F-CAD directly connects to popular machine learning frameworks and takes the developed decoder models as inputs as well as arbitrary hardware budgets and

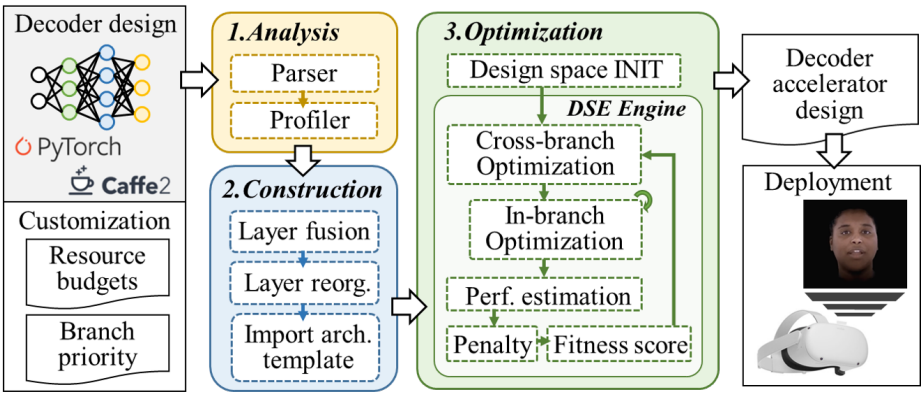

Figure 6.3: The proposed F-CAD design flow with three major steps to deliver optimized hardware accelerators for codec avatar decoding.

differentiated branch priority for more refined customization.

In the *Analysis* step, F-CAD starts analyzing the targeted network by extracting not only layer-wise information (e.g., layer types, layer configurations), but also branch-wise information (e.g., branch number, number of layers in each branch, and layer dependencies). Then, the profiler begins to calculate the compute and memory demands of each layer and provides statistics on branch-wise demands to help map the targeted decoder onto our proposed accelerator architecture. Inputs also contain resource budgets and branch priority to set up resource boundaries and highlight the importance of different branches for architecture exploration in Step 3.

In the *Construction* step, layer fusion is performed to reduce the layer number, where lightweight layers (e.g., activation layers) are aggregated to their neighboring major layers, such as Conv-like and up-sampling layers, which dominate the computation or memory consumption. Branches with shared parts are then separated to create individual dataflows, and the corresponding layers are reorganized and assigned to the flow with the highest computation demand. This strategy helps avoid hardware redundancy, as no duplicated hardware units will be instantiated, and it creates a clear critical flow (with the most computations) from the shared branches and guarantees this flow will get enough attention in the *Optimization* step. For example, Br. 2 and 3 of the targeted decoder share the same front-end. Layers from this part will be assigned to Br. 2 as it is more critical and contains higher computation demand. After fusion and reorganization, F-CAD imports and expands the proposed elastic architecture (Section 6.4) along X and Y dimen-

sions according to the layer and branch number, respectively. Eventually, a basic accelerator is generated and will be optimized in Step 3.

In the *Optimization* step, the accelerator design space (Section 6.5.1) is first determined. The layers and branches of the decoder contribute to the higher dimensional design space, so it becomes complex to search for the optimized design. F-CAD introduces a DSE engine (Section 6.5.2) to leverage both cross-branch and in-branch optimization. A stochastic search is applied in the cross-branch optimization to explore resource distribution schemes across branches; a greedy search is adopted for in-branch optimization, finding the best accelerator candidate for each branch by considering design spaces and available resources. After that, accelerator candidates are evaluated against performance, efficiency, and customization requirements. The DSE engine eventually generates the globally optimized design through an iterative process.

6.4 Accelerator Architecture

6.4.1 A layer-based multi-pipeline accelerator paradigm

The design paradigm of our proposed accelerator is presented in Figure 6.4 (a). Inputs of each branch (e.g., three 256-dimensional latent codes named $In\ a_1 \sim a_3$ for branch 1) are processed in a pipeline manner and passed through all pipeline stages belonging to that branch. For branches with the shared part, corresponding stages are assigned to one of the branches following the layer reorganization strategy. For example, Br. 2 and 3 share the first two layers, so stage $1 \sim 2$ are assigned to Br. 2 in this case, while the subsequent stages are separately executed. The results of stage 2 are distributed to two different branches. We also adopt the fine-grained pipeline design from [10] to lower the pipeline initial latency.

6.4.2 The elastic architecture with 2D expansion capability

To enable the proposed accelerator paradigm, F-CAD introduces an elastic architecture to flexibly expand the accelerator following two dimensions. In Figure 6.4 (b), this elastic architecture consists of basic architecture units

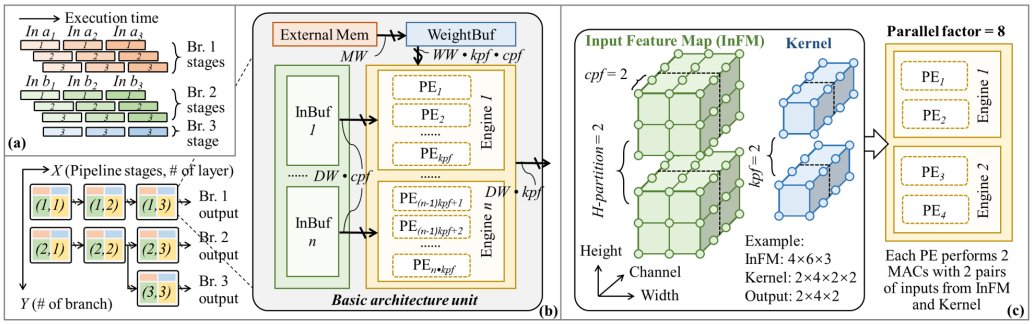

Figure 6.4: (a) F-CAD generated accelerators follow a layer-based multi-pipeline accelerator paradigm, where each layer after reorganization contributes to a pipeline stage; (b) The proposed elastic architecture features a **flexible expansion capability**, which can be extended along X- and Y-axis to instantiate multiple basic architecture units according to the branch number and the layer number in each branch for parallel processing across branches. Each basic architecture unit is equipped with external memory, on-chip memory (for weight buffers and input buffers), and computation resources (for PEs). It provides maximum **three-dimensional parallelism (3D parallelism)** following input channel, output channel, and the height of the input feature map when handling compute-intensive DNN layers; (c) Example of data partitioning to enable the proposed **3D parallelism**.

that are arranged in a two-dimensional plane reflecting the layer reorganization results and each unit is responsible for one pipeline stage. For example, the expansion following the X-axis, such as unit instances (1,1), (1,2), and (1,3), means more stages (three in this case) need to be handled in this branch; the expansion along the Y-axis represents more branches are used in the targeted decoder. In this example, F-CAD generates an accelerator with three pipelines corresponding to Br. $1 \sim 3$.

Inside the basic architecture unit, there are three types of resources: computation (yellow area), on-chip memory (blue and green areas), and external memory (red area) resources. The input feature map from the previous layer is passed horizontally from the left, and a fraction of it is kept in the input buffer (InBuf) to provide a timely data supply. Meanwhile, DNN parameters are fetched from external memory and stored in the weight buffer (WeightBuf) following the computation order. Each basic architecture unit is highly configurable to meet various requirements from different layer stages. It supports the proposed 3D parallelism, which includes two unrolling factors along the output and input channels (kernel parallelism factor kpf and channel

parallelism factor *cpf*) and the partition factor of the input feature map (*H-partition*). After configuration, *H-partition* number of compute engines are instantiated and each engine contains *kpf* PEs to handle computations. The proposed basic architecture unit also allows customized bitwidth of the input features (DW), the weights (WW), and the external memory bus (MW).

6.4.3 The basic architecture unit with 3D parallelism

Figure 6.4 (c) provides a detailed illustration of the proposed 3D parallelism. Assuming a Conv-like layer with a 4×6×3 input feature map (InFM) and two 4×2×2 kernels. We have the maximum input parallel factor as $cpf_{max} = 4$ and the maximum output parallel factor as $kpf_{max} = 2$, since this layer contains four input channels and two output channels that can be processed in parallel. In this case, we configure both input and output parallel factors as 2 ($cpf = kpf = 2$), so each compute engine will instantiate two PEs and each PE performs two multiply-accumulations (MACs) in parallel. Since the parallelism from input/output channels may not be sufficient for codec avatar decoding, we add one additional parallelism by partitioning the InFM along the height dimension. So, all InFM subsections can be processed in parallel. The total parallel factors of this example are $cpf \times kpf \times$ *H-partition* $= 8$ with four PEs instantiated.

6.5 Multi-Branch Design Space Exploration

6.5.1 Multi-branch dynamic design space

Although the highly configurable and scalable features of the proposed elastic architecture help address the unique network structures adopted by codec avatar decoders, they also introduce a complicated and high-dimensional design space. The more branches in the decoder or more layers in a branch there are, the higher dimensional design space it becomes. We define it as a multi-branch dynamic design space and summarize all configurable hardware parameters in Table 6.3. Assuming a decoder with B branches, we pick the first two (with l and m layers) and the last branch (with n layers) as examples. Each of them can be configured in four major aspects as the

Table 6.3: The multi-branch dynamic design space

Br.	**Hardware configurable parameters**
1	$config^1 \leftarrow batchsize^1$, $cpf_1^1 \cdots cpf_l^1$, $kpf_1^1 \cdots kpf_l^1$, $h_1^1 \cdots h_l^1$
2	$config^2 \leftarrow batchsize^2$, $cpf_1^2 \cdots cpf_m^2$, $kpf_1^2 \cdots kpf_m^2$, $h_1^2 \cdots h_m^2$
$\cdots$	$\cdots\cdots$
B	$config^B \leftarrow batchsize^B$, $cpf_1^B \cdots cpf_n^B$, $kpf_1^B \cdots kpf_n^B$, $h_1^B \cdots h_n^B$
Customization	Q, $BatchSize_1 \cdots BatchSize_B$, $P_1 \cdots P_B$
Resource budgets	C_{max}, M_{max}, BW_{max}

cpf, *kpf*, and *H-partition* (h) from the proposed 3D parallelism and batch size (*batchsize*). Parameters of the same branch are passed to a configuration file ($config$), and all these files together describe the overall accelerator configuration. The goal of F-CAD is to explore the best accelerator configuration in the design space by considering the customization and resource constraints. The customization includes the data quantization (Q), the branch-wise targeted batch size ($BatchSize$), and the priority (P) to indicate the different importance of each branch. While the resource budgets specify three major resources as compute resource C_{max}, on-chip memory M_{max}, and external memory access bandwidth BW_{max}.

6.5.2 Design space exploration

To effectively search for the best configuration, the proposed DSE engine adopts a two-step strategy with a cross-branch stochastic search and an in-branch greedy search. It follows the divide and conquer idea to first confirm the resource distribution for every branch and then aim for the best individual branch configuration with given resources.

Cross-branch optimization

In Algorithm 6, the proposed DSE engine first randomly generates $\mathcal{P}$ resource distribution schemes (rd). Each scheme is considered as a candidate, corresponding to a cross-branch resource distribution regarding compute resource C, on-chip memory M, and external memory access bandwidth BW.

Algorithm 6: Cross-branch optimization algorithm

Setup resource budgets: $budget = \{C_{max}, M_{max}, BW_{max}\}$
Setup maximum iteration number: $\mathcal{N}$
Import user customization: $\mathcal{U} = \{BatchSize, Priority\}$, where
  $BatchSize = \{BatchSize_1, \cdots, BatchSize_B\}$,
  $Priority = \{P_1, \cdots, P_B\}$
Randomly initialize RD^0 with $\mathcal{P}$ population: $\{rd_1^0, \cdots, rd_p^0\}$
for *iter in range*($\mathcal{N}$) **do**
  for rd_i^{iter} *in* RD^{iter} **do**
    for br_j *in* $\{br_1, \cdots, br_B\}$ **do**
      $config^j \leftarrow$ **InBranchOptim**$(rd_i^{iter}, \mathcal{U})$ ▷ Algorithm 2
    end
    $Config = \{config^1, \cdots, config^B\}$
    $Perf \leftarrow$ **Eval**$(Config)$ ▷ Evaluate performance
    $fitness \leftarrow$ **S**$(Perf, \mathcal{U})$ − **P**$(Perf)$ ▷ Get fitness score
    rd_i^{best}, $rd_{global}^{best} \leftarrow$ **Update**$(fitness, rd_i^{best}, rd_{global}^{best})$
    if rd_{global}^{best} *has changed* **then**
      $Config_{global}^{best} = Congfig$ ▷ Save the best HW config.
    $rd_i^{iter+1} \leftarrow$ **Evolve**$(rd_i^{iter}, rd_i^{best}, rd_{global}^{best}, budget)$
  end
end
return rd_{global}^{best}, $Config_{global}^{best}$ ▷ Output the global optimal design

In each iteration, rd is then passed to Algorithm 2 for a detailed hardware configuration ($Config$) following the proposed architecture template. With the $Config$, we can evaluate the accelerator performance and calculate the fitness score for every candidate. We build a function **S** to provide a weighted score based on the branch performance $Perf = \{perf_1 \cdots perf_B\}$ and branch priority factor as $\sum_{j=1}^{B} perf_j \times P_j$. We also introduce a penalty term **P** to control the branch-wise performance variance as: $\alpha \times \sigma^2(Perf)$. Then, we calculate the fitness score by subtracting **P** from **S**. A candidate with higher fitness score means it is better than others. We define rd_i^{best} and rd_{global}^{best} to keep track of the local best of each candidate across all iterations and the global best candidates. rd_i^{best} and rd_{global}^{best} can clarify the optimization directions in each iteration, so that each candidate can be evolved iteratively to approach the local and the global best positions by a random distance. By performing such a stochastic search, eventually, Algorithm 6 discovers the global best design by considering the given constraints.

Algorithm 7: In-branch optimization algorithm

Input resource distribution: $rd = \{C,\ M,\ BW\}$
Input user customization from $\mathcal{U}$: $BatchSize$
Initialize $config$ for a l-layer branch: $\{pf_1, \cdots, pf_l\}$
for $layer_k$ *in* $\{layer_1, \cdots, layer_l\}$ **do**
    $op_k \leftarrow$ **GetOP**$(layer_k)$
    $norm\_param_k \leftarrow$ **GetReuse**$(layer_k)$
end
$op_{min} = min(op_1, \cdots, op_l)$
$norm\_bw = \sum_{k=1}^{l}(op_k/op_{min}) \times norm\_param_k \times Freq$
for k *in range(l)* **do**
    $pf_k = \lceil BW/norm\_bw \times (op_k/op_{min}) \rceil$ ▷ Parallelism targets
end
while *True* **do**
    for $layer_k$ *in* $\{layer_1, \cdots, layer_l\}$ **do**
        $cpf_k,\ kpf_k,\ h_k \leftarrow$ **GetPF**$(pf_k,\ layer_k)$
        $c_k,\ m_k,\ bw_k \leftarrow$ **Utilization**$(cpf_k,\ kpf_k,\ h_k)$
    end
    $batchsize = min(C/\sum_{k=1}^{l} c_k, M/\sum_{k=1}^{l} m_k, BW/\sum_{k=1}^{l} bw_k)$
    if $batchsize < BatchSize$ **then**
        $\{pf_1, \cdots pf_k\}/2$
    else
        $batchsize = BatchSize$ **break**
    end
end
$config \leftarrow batchsize,\ \{cpf_1 \cdots cpf_k\},\ \{kpf_1 \cdots kpf_k\},\ \{h_1 \cdots h_k\}$
return $config$ ▷ Output HW config.

In-branch optimization

rd is passed to Algorithm 7 for the best in-branch hardware configuration. Since the proposed accelerator follows an unfolded pipeline architecture, its throughput can be maximized when all pipeline stages are load-balanced with similar latency. To achieve this goal, we capture the layer-wise compute demands (op) and data reuse characteristics ($norm\_param$) to obtain the most optimistic layer-wise parallelism targets (pf) by exhausting the allocated bandwidth resources. After that, a greedy search algorithm is applied to approach hardware configurations ($config$) with the largest level of parallelism under resource constraints. It will converge once the parallelism fails to grow.

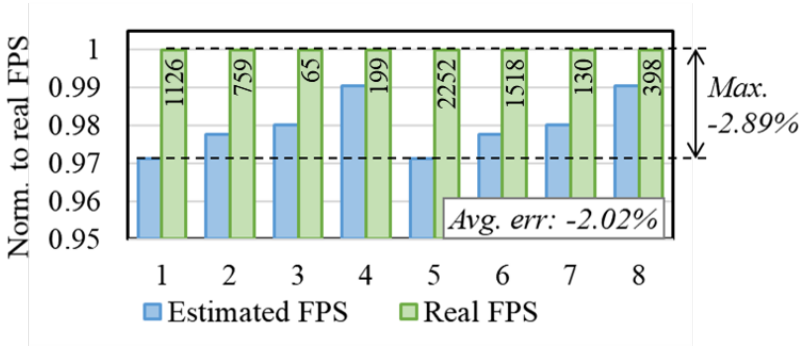

Figure 6.5: F-CAD FPS estimation errors targeting eight benchmarks.

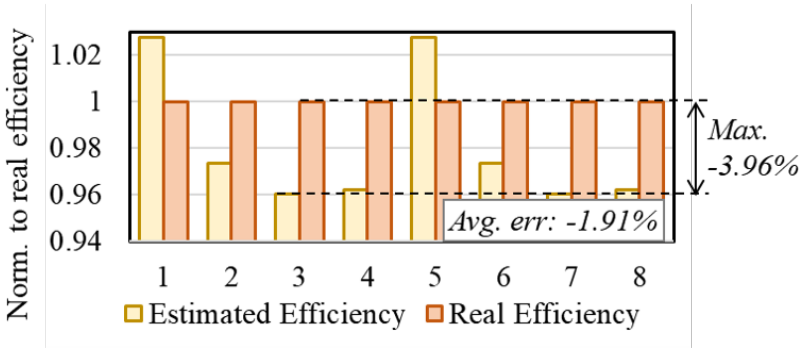

Figure 6.6: F-CAD efficiency estimation errors targeting eight benchmarks.

6.5.3 Performance estimation

We adopt highly accurate analytical models to provide performance and resource utilization feedback and help the DSE engine make the most suitable decisions. Since each branch is individually evaluated, we take a branch with l Conv-like layers as an example. For layer i, we assume the input feature map size $InCh_i \times H_i \times W_i$ and the kernel size $OutCh_i \times InCh_i \times K_i \times K_i$. With hardware configuration $Config$, the latency Lat_i when executing layer i with working frequency f can be determined as:

$$Lat_i = \frac{OutCh_i \times InCh_i \times H_i \times W_i \times K_i \times K_i}{cpf_i \times kpf_i \times h_i \times f} \tag{6.4}$$

The overall throughput (FPS) of this branch is:

$$FPS = \frac{BatchSize}{max(Lat_1, Lat_2, ..., Lat_l)} \tag{6.5}$$

Estimations also include the resource utilization $\{C, M, BW\}$ (by summing up the resource consumed by all layers) and efficiency (by following Equation 6.3). To verify the accuracy of our method, we select DNN benchmarks including AlexNet, ZFNet, VGG16, and Tiny-YOLO with 16-bit (benchmarks $1 \sim 4$) and 8-bit (benchmarks $5 \sim 8$) quantization schemes and compare their estimated performance to the real performance after board-level implementation on a Xilinx KU115 FPGA. As shown in Figure 6.5, we normalize the FPS to the real results in every case to better illustrate the error rate. Real FPS results are also listed in the green bars. The maximum error is only 2.89% while the average error is 2.02%. Similarly, we present the efficiency error in Figure 6.6 with 3.96% maximum error and 1.91% average error.

6.6 Experimental Results

In this section, we target three embedded FPGA platforms (Xilinx Z7045, ZU17EG, and ZU9CG) to demonstrate F-CAD's capability and scalability for accelerating codec avatar decoding. Since the targeted platforms are FPGAs, we set up resource budgets C_{max} and M_{max} as the available DSPs and BRAMs in the targeted FPGA, and BW_{max} as the DDR3 memory bandwidth. The clock frequency is set to 200MHz for all platforms. The targeted decoder is described in Table 6.1 with customized batch size $\{1, 2, 2\}$ corresponding to Br. $1 \sim 3$. Such customization is considered by most VR avatar applications where Br. 2 and 3 need to render two HD textures with specular effects seen by both eyes, while the Br. 1 only outputs one facial geometry that both eyes can share.

Experimental results are listed in Table 6.4, where F-CAD generates five accelerators following the proposed elastic architecture. To evaluate the search speed, we perform 10 independent searches with $\mathcal{N} = 20$ (meaning the search contains 20 iterations) and $\mathcal{P} = 200$ (meaning 200 resource distribution candidates are initialized) for each case, and all of them converge in minutes using an Intel i7 CPU working at 2.6 GHz. The average iteration for convergence is 9.2 (min: 6.8; max: 13.6). Eventually, F-CAD generates optimized designs by considering customization and resource constraints. In particular, the accelerator for case 4 reaches the highest 122.1 FPS, which fully satisfies the VR requirements; while accelerator for case 5

Table 6.4: F-CAD generated accelerators for codec avatar decoding

	Br.	DSP Usage	Total DSPs	BRAM Usage	Total BRAMs	FPS	Efficiency (%)	DSE Time (s)
Case 1: Z7045 (8-bit)	1	199	737 (81.8%)	221	884 (81.1%)	61.0	76.6	101.8
Resource budget:	2	500		551		30.5	86.6	
900 DSPs, 1090 BRAMs	3	38		112		61.0	84.2	
Case 2: ZU17EG (8-bit)	1	351	1357 (83.5%)	280	1024 (64.3%)	**122.1**	86.8	77.3
Resource budget:	2	936		642		61.0	92.6	
1590 DSPs, 1592 BRAMs	3	70		102		**122.1**	91.4	
Case 3: ZU17EG (16-bit)	1	351	1301 (81.8%)	382	1573 (98.8%)	61.0	86.8	82.8
Resource budget:	2	928		983		30.5	93.4	
1590 DSPs, 1592 BRAMs	3	22		208		15.3	72.7	
Case 4: ZU9CG (8-bit)	1	351	2229 (88.5%)	280	1168 (64.0%)	**122.1**	86.8	**56.9**
Resource budget:	2	1808		786		**122.1**	95.8	
2520 DSPs, 1824 BRAMs	3	70		102		**122.1**	91.4	
Case 5: ZU9CG (16-bit)	1	351	2213 (87.8%)	382	1735 (96.1%)	61.0	86.8	67.6
Resource budget:	2	1792		1183		61.0	**96.7**	
2520 DSPs, 1824 BRAMs	3	70		188		61.0	91.4	

Table 6.5: Result comparison to existing customized accelerators (@200MHz)

	DNNBuilder[10]	**HybridDNN**[72]	**F-CAD (our work)**	
Precision	8-bit	16-bit	8-bit	16-bit
DSP	1820	1024	2229	2213
BRAM	1197	1120	1168	1735
FPS	30.5	22.0	**122.1**	**61.0**
Efficiency	28.8%	70.4%	**91.3%**	**91.6%**

delivers the highest efficiency peaking at 96.7%, which can efficiently leverage codec avatar decoding using lightweight HMDs.

We compare F-CAD generated accelerators to existing designs in Table 6.5 by targeting the same ZU9CG FPGA with 2520 DSPs and 1824 BRAMs. We use the same mimic decoder mentioned in Section 6.2.3 for DNNBuilder and HybridDNN, while using the targeted decoder (a real-life decoder) for F-CAD. The batch size is uniformly set to one for fair comparison, as DNNBuilder and HybridDNN do not support differentiated batch schemes. The performance and efficiency of DNNBuilder are limited by insufficient parallelism, so the allocated resources are not fully utilized. On the other hand, HybridDNN fails to allocate more DSPs and leaves more than half of available DSPs unallocated. The reason is that the coarse-grained configuration requires a double-sized accelerator instance to continue scaling, but the BRAM budget is not enough and becomes a bottleneck. In our design, F-CAD delivers the highest FPS and efficiency given the same resource budgets. Compared to DNNBuilder, we achieve 4.0× higher throughput and 62.5% higher efficiency for running the 8-bit codec avatar decoder. Compared to HybridDNN, we can deliver 2.8× higher throughput by allocating only 2.2× more DSPs and 21.2% higher efficiency when running the 16-bit model. F-CAD can also target ASIC designs with the resource budgets $\{C_{max}, M_{max}, BW_{max}\}$ associating to three most commonly used resources in ASIC DNN accelerators: the available MAC units, the on-chip buffer size, and the external memory bandwidth.

6.7 Conclusion

In this chapter, we presented F-CAD, an automation tool to design and explore optimized hardware accelerators for VR avatar decoding with high throughput and efficiency. To address the unique challenges coming from the special DNN structures and demanding performance requirements, we proposed an expandable elastic architecture to support multi-branch DNNs and a highly configurable basic architecture unit to provide flexible and scalable parallel processing. We then introduced a multi-branch dynamic design space to describe hardware configurations and an efficient DSE engine to explore the optimized accelerator by considering various customized constraints and available resource budgets. F-CAD delivered the highest throughput and efficiency, peaking at 122.1 FPS and 91.6%. Compared to the state-of-the-art accelerators, F-CAD achieved 4.0× and 2.8× higher throughput and 62.5% and 21.2% higher efficiency than DNNBuilder and HybridDNN when targeting the same FPGA. F-CAD significantly expands the capability of our toolset proposed in this dissertation, which helps overcome emerging edge AI challenges.

CHAPTER 7

AUTODISTILL: AN END-TO-END FRAMEWORK TO EXPLORE AND DISTILL HARDWARE-EFFICIENT LANGUAGE MODELS

7.1 Introduction

Recently, large-scale pre-trained language models have achieved state-of-the-art results on many tasks. These models not only facilitate a variety of NLP applications but also have continuously improved the result quality of these challenging tasks [26, 118, 119, 52]. Among these models, BERT [52] achieves state-of-the-art performance on a number of NLP tasks and has profoundly affected subsequent model designs [120, 121, 122, 57].

With the advent of such large-scale language models, minimizing the serving cost is becoming increasingly important. They make serving challenging, even for datacenters, due to their sheer sizes. Furthermore, recent advances in techniques to amortize the training cost of large language models, such as fine-tuning, make serving even more costly than training, let alone the fact that training cost is usually amortized over weeks or months while the cost of serving adds up from every request. To alleviate this problem, recent work has extensively investigated model compression techniques, and knowledge distillation is one of the most promising techniques. Earlier knowledge distillation work focuses on distilling large models, like BERT, to task-specific compact designs with less redundancy in model architecture [54, 55, 56], or task-agnostic pre-trained models, which can then be fine-tuned to different downstream tasks [57, 58].

In this chapter, we extend our proposed toolset to support the NLP domain and address two key challenges in practical application of knowledge distillation as part of regular large-scale model release processes: (1) fully automated, efficient distillation process and (2) latency-guided model optimization. Production models in datacenters are diverse and evolve rapidly, which necessitates distillation without human in the loop for scalability. Also,

latency is a crucial metric for production model serving since user-facing products often have strict latency requirements and any latency reduction leads to a significant ownership cost and carbon footprint reduction considering datacenters' large volume. Therefore, we propose AutoDistill, a model distillation framework integrating model architecture exploration and multi-objective optimization for building hardware-aware NLP pre-trained models. To summarize, the main contributions of this paper are as follows.

- We propose an end-to-end framework for fully automated model distillation, which satisfies user-defined metrics and constraints by delivering optimized pre-trained models distilled from large NLP models. It can be easily extended to new search spaces and objectives, thereby eliminating the need for distillation experts. It helps solve the most critical problem of productionizing large-scale model distillation in datacenters and significantly reduces the datacenters' ownership cost and carbon footprint when serving the emerging NLP applications.
- We use Bayesian Optimization (BO) [123, 124] to conduct multi-objective NAS for student model architectures. The proposed search comprehensively considers both prediction accuracy and serving latency on target serving hardware. It is the first time that BO is adopted by the NAS and distillation framework to deliver hardware-efficient large-scale NLP pre-trained models.
- Enabled by AutoDistill, the experiments on TPUv4i identify seven model architectures with up to 3.2% higher pre-trained accuracy and up to 1.44× speedup on latency compared to MobileBERT [57]. Four of them have higher GLUE average scores (up to 81.69) than $\text{BERT}_{\text{BASE}}$ [52], DistillBERT [125], TinyBERT [126], and MobileBERT. Two models are smaller and have higher SQuAD accuracy than DistillBERT, TinyBERT, and NAS-BERT [58].

7.2 Knowledge Distillation Background and Challenges

After BERT was proposed in [52], it has attracted extensive studies on model compression. Knowledge distillation is one widely adopted method to deliver

compact BERT models for serving environments where memory or latency is limited. For example, the authors in [55] perform task-specific knowledge distillation and transfer the knowledge from BERT to a single-layer LSTM model, while the authors in [56] distill smaller BERT models for sequence labeling tasks. Also, a distillation method is developed in [54] to extract knowledge from a teacher model's intermediate and last layers. In addition, DistillBERT [125] performs distillation during model pre-training and reduces the depth of BERT by half. TinyBERT [126] performs layer-wise distillation for model pre-training and fine-tuning. Researchers also focus on building task-agnostic compressed models. For example, MobileBERT can be generically fine-tuned on different downstream NLP tasks [57]. Recent work also shows increasing interest in leveraging NAS for NLP model compression with the goal of discovering more diverse model architectures so that models no longer rely on handcrafted designs [127, 128].

Although knowledge distillation and NAS help diversify the compressed model architectures, existing work mainly focuses on reducing model sizes with accuracy loss as a constraint and formulates model compression as a single-objective optimization problem. Such a strategy may not guarantee that the compressed model can be efficiently deployed on the target hardware, as smaller models do not necessarily perform faster. To address this problem, researchers create a datacenter-optimized network search space and adopt NAS to discover neural networks with optimized accuracy and serving latency [129]. NAS-BERT adopts a look-up table (LUT) to calculate the overall inference latency by adding up costs of the selected operations and uses the latency to guide model search [58]. Similarly, designs in [127] consider the layer-wise hardware costs and calculate the overall model performance as a weighted summation of these costs during the architecture search. However, the weighted summation represents the cost expectation of a group of operations rather than the actual cost of the selected architecture. It also excludes the memory transfer overhead between operations which is very important for accurately modeling performance.

In AutoDistill, we intend to adopt precise hardware feedback by using models' measured hardware performance while running on the target hardware to guide the multi-objective architecture search. Compared to previous designs using proxy or approximated hardware feedback, our method captures more hardware information that cannot be obtained by previous methods.

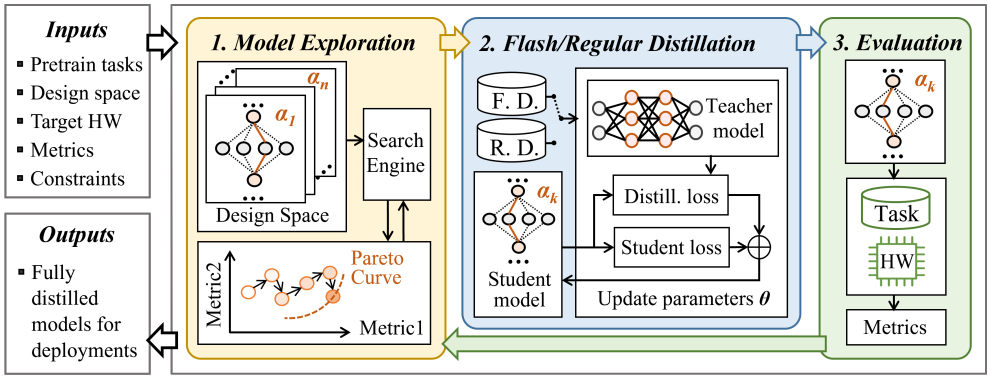

Figure 7.1: Illustration of the proposed AutoDistill framework. *Model Exploration*, *Flash Distillation*, and *Evaluation* are applied iteratively for compressed model exploration. After exploration, the selected models are passed to regular distillation and then output by the framework. We assume n different architectures are defined in the design space and model a_k is proposed in the k-th iteration. F. D. and R. D. represent the training setups for Flash Distillation and regular distillation.

To reduce the search cost, we propose Flash Distillation and adopt the BO algorithm to effectively explore model architectures. All these features are integrated into our proposed framework, AutoDistill, to deliver task-agnostic hardware-efficient pre-trained models.

7.3 The Proposed AutoDistill

7.3.1 Framework overview

AutoDistill provides an end-to-end solution to satisfy user requirements and generates optimized task-agnostic pre-trained models for target hardware. User requirements, including objectives and constraints, are passed to AutoDistill as inputs, which include pre-training tasks, model design spaces, target hardware, evaluation metrics, and constraints (e.g., model size, inference latency limit) that need to be considered. We illustrate the overall flow in Figure 7.1. Three major stages, *Model Exploration*, *Flash Distillation*, and *Evaluation*, are executed in a loop for searching models that best suit the user inputs. Every time the loop is completed, we call it one iteration. After several iterations, the search engine returns the models on the Pareto curve, and they are passed to *Regular Distillation* for more thorough pre-training

so that they can be prepared to serve different downstream tasks.

In *Model Exploration*, the architecture design space is first initialized and passed to the search engine. During every iteration, the engine searches for a better-compressed model by considering the design space, evaluation metrics, and user-specified constraints. We list two metrics as examples in Figure 7.1 (for Metric 1, the higher, the better; while for Metric 2, the lower, the better). After several iterations, found models are plotted on the same coordinate, and AutoDistill selects those located along the Pareto curve as the most promising candidates. We will provide more detailed explanations regarding the design space and the search algorithm design in Section 7.3.2 and 7.5.2, respectively.

AutoDistill then adopts *Flash Distillation* to grow the model recommended by the last stage. This model is considered as a student model, which learns from both pre-training datasets and the teacher model. We include three knowledge distillation technologies: a layer-wise knowledge transfer, a progressive knowledge transfer, and a model pre-training distillation (details in Section 7.4). We have demonstrated that the Flash Distillation needs only 5% of the regular pre-training steps to distinguish promising models at the early stage, which significantly reduces the search efforts. This stage is also responsible for regular distillation with the same teacher model but different training setups (which are the hyperparameters, e.g., training steps, learning rate, and batch size). After iterations of model exploration, regular distillation is launched with more thorough pre-training setups (e.g., more training steps) than Flash Distillation. After that, AutoDistill outputs fully distilled models.

In *Evaluation*, the flash-distilled student model is evaluated with the target tasks and hardware. In general, commonly used metrics include the prediction accuracy (e.g., masked language modeling (MLM) accuracy, next sentence prediction (NSP) accuracy) and the hardware performance (e.g., inference latency, throughput, CE utilization, maximum memory footprint). After all desired metrics are collected, all information is passed to the *Model Exploration* stage, and the search engine selects the next model for the next iteration. We will introduce how we capture the precise hardware performance in Section 7.5.1.

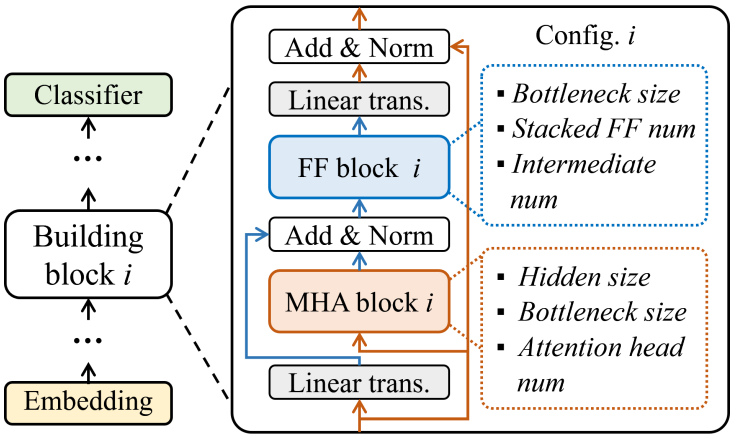

Figure 7.2: The student model design template adopted by AutoDistill for model exploration. It contains a bottleneck structure between two linear transformations and two major configurable blocks including the multi-head attention (MHA) block and the feed-forward (FF) block [57]. The orange and blue arrows represent data flows with different dimensions specified by *Hidden Size* and *Bottleneck Size*.

7.3.2 Model template and architecture design space

To enable more diverse model configurations, we refer to the bottleneck structure from [57] and build a flexible student model template as shown in Figure 7.2. It defines the student model design space by providing multiple configurable network components. With the proposed template, AutoDistill can search for the most suitable model to satisfy user-specific requirements. The proposed template follows a chain structure and consists of a stack of n configurable building blocks, which connect the first embedding layer and the last classifier. It shares a similar block-based network design as BERT to ensure effective knowledge transfer from BERT-like teacher models. For each building block, taking the building block i as an example, there are two major configurable blocks, called multi-head attention (MHA) and feed-forward (FF). These two blocks contribute to a five-dimensional architecture design space that provides a variety of structure combinations.

We summarize the proposed design space in Table 7.1. The first configurable factor is called *Hidden Size*, which indicates the input and output dimensions of the building block and the input dimension of the MHA block. The second factor is *Bottleneck Size*, which shows the output dimension of the MHA block. We have drawn orange and blue lines in Figure 7.2 to re-

Table 7.1: A five-dimensional student model architecture design space. Note that these only serve as examples and the design space can be configured differently based on user requirements.

#	Configurable Factors	Value Choices
1	*Hidden Size*	[128, 246, 384, 512]
2	*Bottleneck Size*	[64, 96, 128, 160]
3	*Attention Head Number*	[1, 2, 4, 8]
4	*Intermediate Number*	[384, 512, 640]
5	*Stacked FF Number*	[2, 4, 6]

spectively denote data flows related to the configuration of hidden size and bottleneck size. By taking advantage of two linear transformation layers, the proposed student model design template can accommodate arbitrary value combinations generated by these configurable factors. Regarding the MHA design, we allow different configurations of the number of heads (*Attention Head Number*). For the FF block design, we consider two configurable factors: *Intermediate Number*) and *Stacked FF Number*. The former explains the FF intermediate size, while the latter illustrates the number of stacked FF networks. Note that every FF network contains two dense layers.

Possible values of these factors are listed in Table 7.1 as examples for quantitative analysis in the following experiments. In total, this example space contains 576 student model design combinations for one building block. Note that AutoDistill supports configurable student model design spaces, which can be further expanded by adding more choices in each configurable factor.

7.4 Flash Distillation

Model accuracy is considered one of the key metrics for evaluating model candidates. That means the ability to quickly determine a model's accuracy potential is crucial to the search efficiency of an end-to-end framework. Therefore, we propose Flash Distillation, a model-agnostic knowledge distillation technique to enable fast selection of promising student models with a great potential for achieving high accuracy. Flash Distillation incorporates

multiple distillation techniques: we adopt layer-wise knowledge transfer for the MHA blocks and the building block feature maps, which shares similar strategies in [54, 125], and we use the progressive knowledge transfer [57] to secure an effective knowledge transfer even for deeper models.

7.4.1 Multiple knowledge transfer schemes

AutoDistill adopts three types of knowledge transfer: MHA transfer $\mathcal{L}_{MHA}$, building block feature map transfer $\mathcal{L}_{FM}$, and the conventional logit transfer during pre-training distillation $\mathcal{L}_D$. Note that AutoDistill is not restricted by the following loss function designs (Equation 7.1 - 7.3) as they can also be configurable by experienced users.

Since the attention mechanism is the unique feature in BERT-like models, we enable MHA knowledge transfer to better guide student models in imitating their teacher's attention block behavior. We adopt Kullback-Leibler divergence (KDL), which is a relative entropy, to measure the differences between the student and the teacher (D_{KDL}), and our goal is to minimize the loss as shown as follows:

$$\mathcal{L}^k_{MHA} = \frac{1}{SH}\sum_{s=1}^{S}\sum_{h=1}^{H} D_{KDL}(a^{\mathcal{T},k}_{s,h}||a^{\mathcal{S},k}_{s,h}), \tag{7.1}$$

where S is the sequence length and H is the number of attention heads. $a^{\mathcal{T}}$ and $a^{\mathcal{S}}$ denote the attention feature maps from the teacher and the student model, respectively. k means the distillation happens in the k-th building block.

Regarding the building block feature map transfer, we minimize the mean squared error between the teacher and the student model, which is shown as follows:

$$\mathcal{L}^k_{FM} = \frac{1}{SN}\sum_{s=1}^{S}\sum_{i=1}^{N}(f^{\mathcal{T},k}_{s,i} - f^{\mathcal{S},k}_{s,i})^2, \tag{7.2}$$

where $f^{\mathcal{T}}$ is the teacher's feature map and $f^{\mathcal{S}}$ is the student's feature map. N denotes the feature map size.

After performing layer-wise knowledge transfers, we include model pre-

training distillation and use $\mathcal{L}_D$ to represent the distillation loss:

$$\mathcal{L}_D = \alpha\mathcal{L}_M + (1-\alpha)\mathcal{L}_{MD} + \mathcal{L}_N, \tag{7.3}$$

where $\mathcal{L}_M$ and $\mathcal{L}_{MD}$ denote the Masked Language Modeling (MLM) loss and the MLM distillation loss. $\mathcal{L}_N$ is the Next Sentence Prediction (NSP) loss. α is a hyperparameter between 0 and 1.

7.4.2 Knowledge transfer with different layer dimensions

Previous work assumes that the teacher and student layers being distilled share the same layer dimension sizes as they are handcrafted with careful considerations of model architectures. However, it is not a reasonable assumption in AutoDistill, since we need to target a much broader design space with arbitrary student model architectures. The layer dimension size of the student model may not always be the same as that of the teacher. To address this issue, we insert a dense layer between the teacher and the student layers where knowledge transfer occurs. For example, if the dimension of the student building block output is not the same as the teacher's, a dense layer is inserted between them. With these additional dense layers, student layers that need to be distilled can be scaled up to match the size of the corresponding teacher layers, so that teacher's knowledge can be transferred smoothly. We have seen a similar solution adopted by [130], using an additional convolution layer to match the feature map size. Layer dimension mismatch also happens between the embedding layer and the first building block because the embedding layer is directly copied from the teacher model. To solve this problem, we down- or up-sample the embedding weights to match the student layer size.

7.4.3 Progressive knowledge transfer

With the mismatch problem solved, we launch the progressive knowledge transfer to help student models quickly acquire knowledge from the teacher model, which was proposed by [57]. Assuming a student model with K building blocks, layer-wise transfers for MHA and building block feature maps are conducted one block after another. It is a K-stage process following the order

from block 1 to K. When working on the k-th stage, all trainable parameters in stage 1 to stage $k-1$ are frozen, so the student model can learn knowledge progressively one building block after another. Next, we continue the model pre-training distillation until the training reaches the preset training step. In addition, we set a fixed ratio between the step number in progressive distillation and those in the pre-training distillation to guarantee that all three types of distillation schemes mentioned in Section 7.4.1 can be applied effectively.

7.4.4 Flash Distillation vs. regular distillation

In the proposed Flash Distillation, the training step count is set to a much smaller number compared to the regular distillation because they are designed for different goals. Flash Distillation works for the early selection of promising student model architectures. It is not necessary to fully train a model to start evaluating it and making decisions. In contrast, regular distillation works for fully preparing student models and make them ready for use. It generally contains hundreds of thousands of steps. We will have more discussions in the experiment (Section 7.6.7) to illustrate how we select the step number for Flash Distillation.

7.5 Hardware-Aware Model Selection

7.5.1 Hardware Performance Integration

Each model is measured with multiple runs for forwarding propagation. In each run, the hardware traces are collected, including inference latency, throughput, CE utilization, as well as memory capacity and bandwidth utilization. Particularly, our experiments use the average serving latency as the desired metric. It can easily be extended to incorporate more metrics in the search engine because the BO algorithm is very flexible for optimizing more objectives. The collected metrics are then passed to *Model Exploration* for guiding the search process. Since the hardware performance evaluation is independent of Flash Distillation or pre-training accuracy, it can be completed in parallel to or offline during the distillation process.

7.5.2 NAS for student models

AutoDistill adopts a search-based solution to explore all possible student model configurations. One major goal is to reduce the reliance on prior knowledge summarized by human experts, so AutoDistill can find designs that have not been discovered before but may achieve better hardware and software performances given user-specific requirements. AutoDistill formulates the student model architecture search as a black-box optimization problem, as it needs minimal assumptions about the problem and minimal internal information of the system [131, 132], i.e., the inner relationship between the selected model architecture and its performance objectives. To be more specific, AutoDistill optimizes the following problem

$$\text{maximize } f(x) : X \to \mathbb{R}^o, \quad s.t. \quad x \in X \tag{7.4}$$

where x represents a set of configurable factors for describing model architectures and o is the number of objectives. In our experiments, o is two, representing accuracy and latency objectives. AutoDistill can be easily customized for more and different objectives. With this formulation, all we need is to choose samples (e.g., any $x \in X$) and evaluate $f(x)$, without needing to access other information.

To solve this problem, AutoDistill leverages Bayesian Optimization (BO), an effective black-box optimization algorithm that does not assume any functional forms of objective problems. It uses Gaussian Processes to learn the posterior distribution of the objective function, which is then used to construct an acquisition function to determine the next trial [124]. We use the BO algorithm implemented in Vizier [123], a cloud-based black-box optimization service, and integrate it into the search engine for student architecture search. Results in Section 7.3.2 show that BO outperforms other algorithms that support multi-objective optimization, including random search and evolutionary algorithms.

In previous designs, the gradient-based method is adopted to speed up NAS (referred as differentiable NAS, or DNAS), and it has been demonstrated to work well by generating more accurate models for computer vision tasks [133, 81]. It is difficult for DNAS to handle NLP tasks, as it is required to train a huge supernet, containing all possible architecture candi-

dates, on the NLP pre-training tasks, which has been proven to be extremely costly [58, 134]. Compared to the DNAS approach, our solution has the following major benefits: 1) AutoDistill does not need to spend enormous effort to train a large supernet beforehand on NLP pre-training tasks; 2) it can better scale to handle a much larger design space; and 3) it can be easily extended to new objectives and new models with different architecture configurations.

7.6 Experimental Results

In this section, we evaluate AutoDistill and demonstrate its effectiveness for exploring hardware-efficient task-agnostic NLP models. First, we present the compressed task-agnostic models found by AutoDistill with optimized pre-training accuracy and hardware performance. Since the search is enabled by the BO algorithm, we compare it to random and evolutionary algorithms to demonstrate its better search efficiency. Next, we evaluate the proposed models on the GLUE benchmark, and we also fine-tune these models on a downstream task called SQuAD and compare them to the state-of-the-art distilled BERT models. We also provide quantitative analysis to show the importance of using multi-objective search and the effectiveness of Flash Distillation.

7.6.1 Experimental Setup

To launch AutoDistill, we first specify the framework inputs as described in Figure 7.1, which include pre-training tasks, design space, target hardware, and target metrics. AutoDistill can also support user-defined constraints to help reduce the search space, such as specifying the minimum acceptable hardware performance. In this experiment, we do not provide any constraints to limit the model exploration. To demonstrate AutoDistill's effectiveness on compressing task-agnostic models, we measure student models' pre-training accuracy [52], which includes MLM and NSP. We evaluate AutoDistill on the design space described in Table 7.1 for building block search and stack 24 of the same building blocks to construct every student model. In this experiment, TPUv4i [135] is used as the target hardware to perform model

inference. Inference latency is measured with batch size = 1, which can be changed to a representative production serving batch size of the user's target model. Since our goal is to deliver hardware-efficient models with high accuracy, we pass both hardware and software metrics to AutoDistill as optimization objectives, which are precise model inference latency and model pre-training accuracy.

In each iteration, one student model is selected by the *Model Exploration* stage and passed to the *Flash Distillation* stage for rapid model pre-training. The teacher model is called IB-BERT$_{\text{LARGE}}$, which is a 24-layer BERT-like model with 293M parameters proposed by [57]. The selected student model follows the pre-training schedule introduced in Section 7.4, which includes the layer-wise knowledge transfer, the progressive knowledge transfer, and the model pre-training distillation. Each building block is quickly trained by 500 steps for layer-wise knowledge transfer, so the progressive knowledge transfer lasts for 500×24=12k steps. Next, the student model is pre-trained for 25k steps with 500 warm-up steps. In total, a Flash Distillation contains 37k steps and it is performed on TPU v3 [1] chips with a batch size of 2048 and LAMB optimizer [136]. For pre-training data, we follow the same recipe as BERT, using the BooksCorpus [137] and English Wikipedia.

7.6.2 Models that outperform the state-of-the-art

MobileBERT [57], the state-of-the-art task-agnostic design for BERT compression, is selected as our baseline. To ensure a fair comparison, we run the open-sourced MobileBERT code [2] and use the same hyperparameters for MobileBERT pre-training and our models' regular distillation. All models are trained from scratch.

After Flash Distillation, AutoDistill finds seven compressed models that exhibit higher pre-training accuracy and lower inference latency on the target hardware than MobileBERT with the same Flash Distillation process. We then run regular distillation for these seven models for a thorough training process. It is a much longer training process, which includes 240k-step progressive knowledge transfer (10K steps for each layer) and 500k-step model pre-training with 10k warm-up steps. All models are trained on TPU v3

[1] https://cloud.google.com/tpu

[2] https://github.com/google-research/google-research/tree/master/mobilebert

Table 7.2: The student models found by Autodistill. We train these models using the same pre-training setup as the baseline (with 740k-step pre-training and the batch size of 2048) and present their pre-training accuracy (MLM accuracy) and measured inference latency results. All models achieve better accuracy and lower latency compared to the baseline. The number shown in parentheses is the result of comparison to the baseline. ‡denotes our runs with the open-sourced MobileBERT code. For reference, MLPerf [138] uses MLM accuracy = 72 as the target pre-training accuracy of $\text{BERT}_{\text{LARGE}}$.

	# Param	Latency	Accuracy
MobileBERT‡(baseline)	25.3 M	0.65 ms	69.2
Model_512_128_1_384_4	22.2 M (87.7%)	0.54 ms (1.21×)	70.0 (101.3%)
Model_512_128_1_640_2	20.6 M (81.4%)	**0.45 ms** (1.44×)	69.5 (100.4%)
Model_512_128_1_640_4	28.5 M (112.6%)	0.58 ms (1.12×)	**71.4** (103.2%)
Model_512_128_2_640_2	20.6 M (81.4%)	0.49 ms (1.32×)	70.0 (101.2%)
Model_512_128_4_512_2	**19.0 M** (75.1%)	0.54 ms (1.21×)	69.7 (100.8%)
Model_512_128_4_640_2	20.6 M (81.4%)	0.56 ms (1.16×)	70.3 (101.6%)
Model_512_160_2_512_2	22.8 M (90.1%)	0.59 ms (1.09×)	70.4 (101.8%)

chips with a batch size of 2048.

After regular distillation, we list their accuracy and inference latency in Table 7.2. The compressed model candidates are encoded with their five architecture configurable factors described in Table 7.1. For example, the model name Model_512_128_1_384_4 encodes a model architecture with *Hidden Size*=512, *Bottleneck Size*=128, *Attention Head Number*=1, *Intermediate Number*=384, and *Stacked FF Number*=4.

All seven models outperform the baseline regarding both accuracy and inference latency. Among them, Model_512_128_1_640_4 achieves the best accuracy, peaking at 71.4. That is 3.2% higher than the baseline. The other one, Model_512_128_1_640_2, maintains a competitive accuracy while achieving 1.44× speedup on inference latency compared to the baseline. Model_512_128_4_512_2 achieves the smallest model size with 19.0M parameters (75.1% of the baseline). We observe that models with more parameters do not always have longer inference latency: Model_512_128_1_640_4 has 12.6% more parameters, but it performs 1.12× faster than the baseline.

Such a counter-intuitive observation could be caused by the larger model's better operational intensity and parallelism, or that its computation pattern

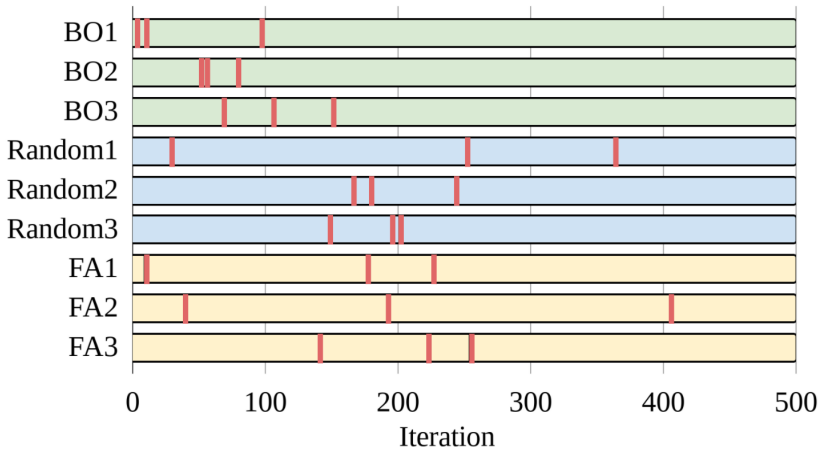

Figure 7.3: Search efficiency of Bayesian optimization (BO), random search, and firefly algorithm (FA). The three red vertical lines in each bar indicate the iteration numbers where three better-than-MobileBERT student models are found. BO finds three models with fewer iterations than other two algorithms.

is better optimized by the software stack. Similar findings are published in [129], where models with more parameters can achieve better latency than those with fewer parameters. This indicates that optimizing for model size during distillation is not sufficient for better serving latency. This finding also emphasizes the importance of using precise hardware performance instead of proxy metrics to guide the model search.

7.6.3 Search Algorithm Comparison

The search engine in the *Model Exploration* stage is one of the most crucial components facilitating the search efficiency of AutoDistill. An efficient search algorithm that gives good candidates with fewer iterations during the model search can reduce the computation cost and end-to-end duration of NAS. To better understand its search efficiency, we compare the BO algorithm used in AutoDistill to two other popular algorithms with multi-objective support: the random search algorithm and the firefly algorithm (FA) [139]. The random search represents a naive baseline, which is commonly compared against in previous work [134]. In each iteration, a random search selects one of the possible models uniformly at random, and every trial is independent of other trials. The FA represents a more sophisticated

baseline, and it generates a new suggestion every iteration by taking a linear combination of previous suggestions with a small perturbation.

In this experiment, we measure the number of NAS iterations these search algorithms take to search for three pre-trained models that outperform the baseline model (MobileBERT). For a fair comparison, we maintain the same setup, including the design space, the metrics, and the Flash Distillation technique, while evaluating different search algorithms. In AutoDistill, the search engine proposes one model candidate per iteration, by going through the three major stages of *Model Exploration*, *Flash Distillation*, and *Evaluation*. We set the maximum iteration number as 500 to provide sufficient time for these algorithms and evaluate how many iterations they take to find three models outperforming MobileBERT.

Comparison results are shown in Figure 7.3, where the X-axis represents the number of iterations and the Y-axis indicates nine independent experiments (three trials per algorithm). The results show that BO can finish searching much earlier than the other two algorithms. For the best case, it discovers all three models within 80 iterations, and, on average, it can finish in 110 iterations. As a comparison, the random algorithm and the FA require 270 and 296 iterations on average to find all three better models. The average number of iterations to discover the first promising model is another very important metric to compare, because in production, people usually need only one model to deploy. BO takes 43 iterations on average to find the first promising model, while it is 116 and 64 respectively for random algorithm and FA.

7.6.4 Results on GLUE

We demonstrate the AutoDistill generated models on the General Language Understanding Evaluation (GLUE) benchmark with nine downstream natural language understanding tasks [140]. As shown in Table 7.3, we compare our pre-trained models from Table 7.2 to $\text{BERT}_{\text{BASE}}$ [52] and state-of-the-art BERT compression models, including DistillBERT [125], TinyBERT [126], NAS-BERT [58], and MobileBERT [57].

All of the AutoDistill generated models in Table 7.3 achieve higher average scores than $\text{BERT}_{\text{BASE}}$, DistilBERT, TinyBERT_6, and MobileBERT with

Table 7.3: The results on the GLUE benchmark. *denotes models conducting knowledge distillation in both pre-training and fine-tuning stages. ∘marks MobileBERT without operational optimizations. ▷marks NAS-BERT with data augmentation.

	# Param	Latency	CoLA	MNLI-m/mm	MRPC	QNLI	QQP	RTE	SST-2	STS-B	Avg.
$\text{BERT}_{\text{BASE}}$	109 M	-	52.1	84.6/83.4	**88.9**	90.5	71.2	66.4	**93.5**	85.8	79.60
DistilBERT	67 M	-	51.3	82.2	87.5	89.2	88.5	59.9	91.3	86.9	79.60
TinyBERT_6*	67 M	-	51.1	84.6/83.2	87.3	90.4	71.6	70.0	93.1	83.7	79.44
NAS-BERT*	60 M	-	48.4	83.5	84.5	90.9	88.9	**73.7**	92.9	86.1	81.11
NAS-BERT*▷	60 M	-	50.5	84.1	86.4	91.2	88.8	72.7	92.6	86.9	81.65
MobileBERT	25.3 M	0.65 ms	50.5	83.3/82.6	88.8	90.6	70.2	66.2	92.8	84.4	78.82
MobileBERT∘	25.3 M	0.65 ms	51.1	84.3/83.4	88.8	**91.6**	70.5	70.4	92.6	84.8	79.72
Model_512_128_1_640_2	**20.6 M**	**0.45 ms**	53.2	81.0/81.9	84.1	88.9	89.4	67.2	90.8	87.0	80.38
Model_512_128_2_640_2	**20.6 M**	0.49 ms	53.2	82.3/82.5	83.8	90.2	89.7	66.1	90.8	88.1	80.75
Model_512_160_2_512_2	22.8 M	0.59 ms	52.1	82.6/82.9	86.3	90.4	90.0	63.9	91.2	88.7	80.89
Model_512_128_1_640_4	28.5 M	0.58 ms	**55.9**	82.7/82.8	87.5	90.4	**90.2**	66.1	90.8	**88.8**	**81.69**

significantly smaller model sizes. Among them, two of our most compact models (Model_512_128_1_640_2 and Model_512_128_2_640_2) have a 81.1% size reduction compared to $\text{BERT}_{\text{BASE}}$ and deliver higher average scores. Our most accurate model (Model_512_128_1_640_4) achieves the highest scores in three tasks (CoLA, QQP, and STS-B) and the highest average score (81.69) across all nine tasks. It is worth mentioning that AutoDistill is task-agnostic and does not require teacher models (no distillation) for downstream tasks, while TinyBERT and NAS-BERT require teacher models in fine-tuning which have unfair advantages over AutoDistill.

7.6.5 Results on SQuAD

To further evaluate the model quality that AutoDistill found, we assess the pre-trained models from Table 7.2 with a downstream NLP task called Stanford Question Answering Dataset (SQuAD) [141]. It is a large-scale dataset with 100k crowd-sourced question/answer pairs for question answering and reading comprehension. We choose the dev F1 and the exact match (EM) as accuracy metrics.

Table 7.4 compares our models to the $\text{BERT}_{\text{BASE}}$ [52] and the recently published compressed designs, including DistillBERT [125], TinyBERT [126], NAS-BERT [58], and MobileBERT [57]. Note that DistillBERT performs knowledge distillation only in model pre-training stage, while DistillBERT* (distinguished with *) uses two-stage knowledge distillation for both model pre-training and fine-tuning. The TinyBERT_6* denotes the 6-layer model with two-stage distillation and it is the most accurate model proposed in [126]. Similarly, both NAS-BERT designs do distillation in both stages and NAS-BERT*†uses 1.6× more pre-training steps than NAS-BERT*. Such a two-stage distillation strategy generally involves a more complicated distillation pipeline as it needs additional fine-tuned teachers for different downstream tasks.

In contrast to the above-mentioned designs, MobileBERT and our solutions only require a single-stage knowledge distillation during pre-training, so that the pre-trained models can be directly fine-tuned for downstream tasks. We collect MobileBERT's performance from [57], which involves hyperparameter tuning, especially for the SQuAD task to achieve better accuracy. In

Table 7.4: The results on SQuAD v1.1. *denotes models conducting knowledge distillation in both pre-training and fine-tuning stages. †denotes a model using 1.6× more pre-training steps than its original setup. ‡marks our run with the open-sourced MobileBERT code without hyperparameter tuning and using the same fine-tuning setup as our models.

	# Param	Latency	F1	EM
$\text{BERT}_{\text{BASE}}$	109 M	-	88.5	80.8
DistilBERT	67 M	-	85.8	77.1
DistilBERT*	67 M	-	86.9	79.1
TinyBERT_6*	67 M	-	87.5	79.7
NAS-BERT*	60 M	-	88.0	80.5
NAS-BERT*†	60 M	-	88.4	81.2
MobileBERT	25.3 M	0.65 ms	**90.0**	**82.9**
MobileBERT‡	25.3 M	0.65 ms	87.7	80.0
Ours-1	22.8 M	0.59 ms	88.4	80.8
Ours-2	**20.6 M**	**0.49 ms**	88.1	80.5

addition, we run its open-sourced code following our setups in Section 7.6.1 without hyperparameter tuning and present the results with the mark ‡. The reason why MobileBERT‡'s results are different from MobileBERT [57] is that we do not conduct hyperparameter tuning as the original paper does to make it a fair comparison for the rest of the models.

Table 7.4 shows that the student models found by AutoDistill have lower inference latency and smaller model sizes while maintaining great accuracy for the SQuAD task. The average F1 and EM of the seven models are 88 and 80, respectively. Among them, the most accurate one (Ours-1: Model_512_160_2_512_2) achieves the same F1 score (88.4) with only 38% of the parameters compared to NAS-BERT*†. The more efficient model (Ours-2: Model_512_128_2_640_2) has a more compact architecture with only 20.6M parameters and higher accuracy than five other compressed models listed in Table 7.4.

7.6.6 Multi-objective vs. single-objective search

Additionally, we compare the multi-objective and the single-objective search and show their different effects on compressed model exploration. The multi-

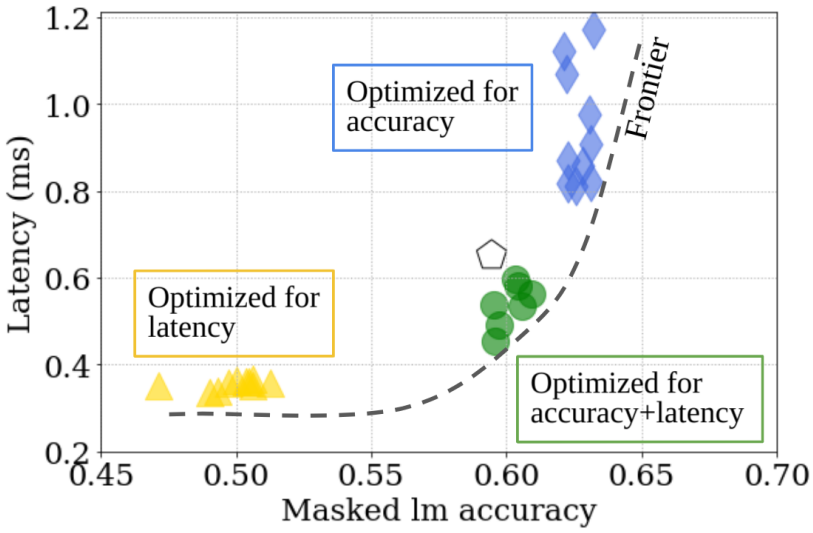

Figure 7.4: Multi-objective search in AutoDistill helps identify the most promising models by considering both hardware and software metrics after Flash Distillation. The pentagon marks the performance of the MobileBERT representing the baseline. The clusters with yellow triangles and blue diamonds show the models optimized for latency and accuracy, respectively. The green dots at the bottom right are the seven models listed in Table 7.2, which are discovered by multi-objective search in AutoDistill.

objective search experiment maximizes accuracy and minimizes latency. And the two experiments of single-objective search respectively optimize for model accuracy (which is more popular in previous work) and serving latency. The same model architecture design space in Table 7.1 is used for all three search strategies.

Figure 7.4 shows the most promising models found, with accuracy shown on the X-axis and latency on the Y-axis. We observe totally different behaviors from these search objectives. The single-objective approaches output models that are located close to the two ends of the Pareto curve: one along the lower latency limit (denoted as yellow triangles for optimized latency) and the other along the upper accuracy limit (denoted as blue diamonds for optimized accuracy). However, these models may not be ideal for production deployment which usually needs to accommodate more than one optimization objective, such as a better balance between accuracy and latency. By considering multiple objectives, the search can find all models near the Pareto curve. Assuming users are interested in models with both high accuracy and low latency, they can select models toward the middle of the Pareto curve

with all objectives being optimized. In our experiments, those are models (denoted as green dots) near the lower right. Compared to models that only focus on higher accuracy (the blue ones), these models (the green ones) have 1.8× speedup on average with a loss 2.5% accuracy, which could be valuable for production deployment and would not have been found with a single-objective search. It is important for the multi-objective search to find all models along the Pareto curve so that users can have a better idea about the design space and select ones that are of interest by setting different weights for the objectives.

Besides the unconstrained search introduced in the last paragraph, another common single-objective search approach considers metric constraints, for example, maximizing accuracy with a latency constraint or vice versa. However, this approach needs reasonable constraints to find models similar to what can be found by multi-objective search, and setting constraints is difficult without prior knowledge of the Pareto curve. Let us use the results in Figure 7.4 as examples and assume a single-objective search tries to maximize accuracy with a latency constraint. If the latency constraint is lower than 1ms, the search results are similar to optimizing for accuracy only, which are some of the blue diamonds. If the latency constraint is lower than 0.4ms, we will get models with worse accuracy than the baseline. It is much easier to adopt a multi-objective search to find the student models (in green dots) which are better than the baseline.

7.6.7 Flash Distillation

With Flash Distillation, AutoDistill is able to select models with large potential to achieve high accuracy at an early pre-training stage. We evaluate the accuracy potential of student models with a smaller number of training steps during knowledge transfer and skip unpromising candidates for regular distillation.

To determine a suitable step number, we launch four Flash Distillation tests with 1.25%, 2.5%, 5%, and 10% of the step number in the regular distillation (which contains 740k steps). The ratio between progressive and pre-training distillation is set to 0.48 for all cases. We plot the pre-training accuracy and distillation time for each case in Figure 7.5. Since the accu-

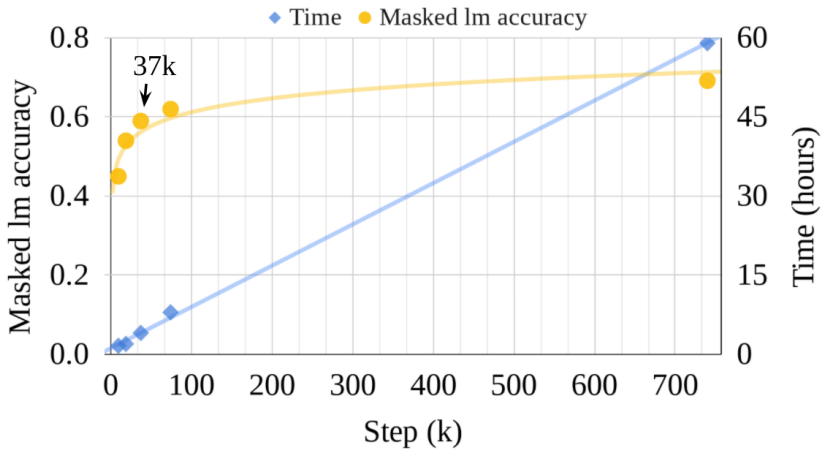

Figure 7.5: Illustration of four Flash Distillation cases with 9.25k, 18.5k, 37k, and 74k steps comparing to the regular distillation case with 740k steps for the same student model (a total of five data points). The growth of accuracy shows a logarithmic trend while the time spent on training increases linearly with increasing training steps.

racy shows a logarithmic trend, selecting a spot when the accuracy growth becomes slower seems a great trade-off between training costs and accuracy gain. We, therefore, select 37k (5% of the regular distillation steps) as the Flash Distillation step number.

To verify that Flash Distillation is effective in selecting promising candidates, we perform flash and regular distillation for models from Table 7.2 and compare the changes in their relative positions in accuracy and latency dimensions. Figure 7.6 shows the results of the same group of models being pre-trained for 37k (in Flash Distillation) and 740k steps (in regular distillation), respectively. Each dot represents one student model in Table 7.2 and the pentagon denotes the baseline model (MobileBERT). Since different distillation methods for the same model only affect model accuracy, we observe that dots are shifted to the right, meaning they become more accurate after a much longer pre-training process. Their relative positions, however, are fairly stable regardless of using Flash Distillation or regular distillation. In this experiment, Flash Distillation only consumes 5% of the training steps to grow the models compared to using regular distillation and models can already be distinguished even though they have not reached the final accuracy. We therefore conclude that Flash Distillation is helpful in facilitating the early selection of promising models.

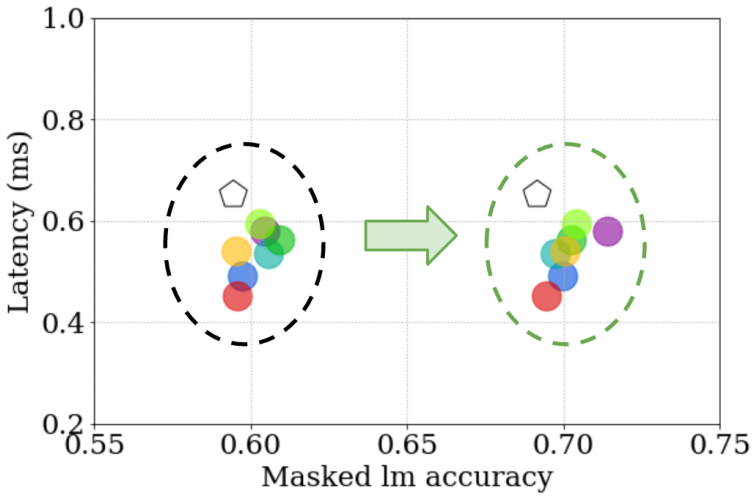

Figure 7.6: The models selected using Flash Distillation with 37k steps (the left cluster) and the same ones using regular distillation with 740k steps (the right cluster). Promising models can be identified much earlier using Flash Distillation. Note that the pentagon denotes the performance of MobileBERT (baseline).

7.7 Conclusion

This chapter presented AutoDistill, an end-to-end model distillation framework that integrates model architecture exploration and multi-objective optimization to explore hardware-efficient task-agnostic NLP models for datacenter deployment. We formulated the compressed model exploration as a black-box optimization problem and adopted BO to conduct a multi-objective model architecture search. To rapidly identify more promising models, we introduced a model-agnostic pre-training technique called Flash Distillation to enable fast knowledge distillation for compressed model candidates. Experiments on TPUv4i showed that models generated by AutoDistill achieved up to 3.2% higher pre-trained accuracy and up to 1.44× speedup on latency compared to MobileBERT [57]. By evaluating on GLUE, our most accurate model with 28.5M parameters achieved an 81.69 average score, which outperformed $\text{BERT}_{\text{BASE}}$ [52], DistillBERT [125], TinyBERT [126], NAS-BERT [58], and MobileBERT. Our most compact model with 20.6M parameters (81.1% size reduction compared to $\text{BERT}_{\text{BASE}}$) still achieved a higher average score than $\text{BERT}_{\text{BASE}}$, DistillBERT, TinyBERT, and MobileBERT. By evaluating on SQuAD, two of the proposed models with less than 23M parameters achieved higher accuracy than DistillBERT, TinyBERT, and NAS-

BERT. Results demonstrated that the proposed AutoDistill delivered more efficient NLP models, greatly reducing ownership cost and carbon footprint during high-quality NLP model serving on datacenters.

AutoDistill successfully expands the applicability of our proposed toolset to cover NLP domains by delivering state-of-the-art NLP solutions. Its integration of model distillation and network architecture search enables more efficient design space exploration, model training, and model selection, effectively addressing the challenges of the latest NLP model development. With AutoDistill, our toolset provides comprehensive coverage of both computer vision and NLP applications.

CHAPTER 8

CONCLUSION

The recent development of AI applications causes significant challenges for hardware deployment, as they require not only high inference accuracy but also high inference speed, throughput, and energy efficiency. Although AI hardware acceleration can alleviate these problems, design difficulties still come from limited hardware resources, restricted power budgets, tedious hardware design, intricate hardware verification problems, and time-consuming accelerator design space explorations.

To address these challenges, this dissertation introduced a comprehensive toolset to deliver efficient AI hardware acceleration for various real-life scenarios. The proposed tools include HLS-based DNN accelerator design and optimization strategies, end-to-end automation tools for customized accelerator designs, and DNN-accelerator co-design and co-optimization strategies. With these proposed tools, we generated state-of-the-art solutions to accelerate popular AI applications, covering image/video understanding in Chapter 3, image classification in Chapter 4, object detection and tracking in Chapter 5, VR codec avatar in Chapter 6, and various NLP tasks in Chapter 7.

We first demonstrated the proposed HLS-based design flow by delivering an LRCN accelerator for real-time video analysis. We successfully addressed the design challenges in managing computational complexity, on-chip memory limitation, and external memory bottleneck. This work contains three novel designs: highly optimized IPs as the accelerator's building blocks to speed up computation; an efficient hierarchical memory system to alleviate slow data transfer; and multiple technologies (e.g., network pruning, quantization) to reduce model size while maintaining model accuracy. We also introduced a resource allocation strategy called REALM to drive theoretical guidelines for optimized accelerator configurations to achieve minimal execution latency. The proposed accelerator showed 4.8× and 3.1× speedup compared to an Intel Xeon CPU and an NVIDIA K80 GPU, while it consumed 17.5× less

energy for every image processed. It is the first FPGA-based LRCN accelerator to enable real-time image captioning with careful considerations of the model accuracy and energy efficiency. This work is open-source, which provides successful design templates to address challenges in managing compute- and communication-intensive AI workloads.

We then introduced DNNBuilder to address the challenges caused by inefficient hardware accelerator design and optimization. These challenges create an ever-widening barrier between fast AI model design in software and slow hardware accelerator implementation. To bridge the hardware-software gap, we proposed DNNBuilder, an integrated design flow for building FPGA-based AI accelerators automatically from popular machine learning frameworks. Users are no longer required to design and optimize accelerators manually but can enjoy the auto-generated hardware accelerators for desired AI workloads. We introduced two major architecture innovations, the fine-grained layer-based pipeline architecture and the column-based cache scheme, which achieved 7.7× and 43× reduction in latency and on-chip memory usage. To adopt the diverse demands of edge- and cloud-computing, we proposed an automatic design space exploration tool to generate optimized architecture configurations by considering the targeted AI workload and available hardware resources. DNNBuilder is an open-source project to make high-quality AI hardware accelerators easier to implement. The generated accelerators can be adapted to various AI applications and provide real-time and high throughput AI services. Because of the state-of-the-art accelerator designs, DNNBuilder has been awarded the ICCAD William J. McCalla Best Paper Award, and it has been widely adopted by the industry.

Next, we introduced SkyNet to provide DNN-accelerator co-design and co-optimization strategies. Instead of following the standard top-down compact DNN design flow, we proposed a novel bottom-up design approach to construct DNNs from basic building blocks, which can capture comprehensive hardware constraints. An evolutionary algorithm was introduced to evolve the network candidates toward higher accuracy and efficiency. We demonstrated the effectiveness of SkyNet by winning a competitive System Design Contest at DAC-SDC for low-power object detection. It outperformed 100+ competitors and delivered the best solutions for embedded GPUs and FPGAs. For object tracking, SkyNet enabled 1.6× and 1.7× faster throughput and the same accurate outputs by replacing the ResNet-50 backbone in two

popular trackers named SiamRPN++ and SiamMask.

Following the co-design strategies, we proposed an efficient method, called F-CAD, to deliver customized accelerators for emerging VR applications running on extremely lightweight edge devices. The F-CAD generated designs with up to 4.0× higher throughput and up to 62.5% higher energy efficiency than state-of-the-art designs and which perfectly met the demanding VR requirements. Additionally, we proposed AutoDistill to address the difficulties of serving large-scale NLP models in the cloud. It is the first fully automated framework that integrates model distillation and neural architecture search to deliver hardware-efficient NLP pre-trained models. Compared to the state-of-the-art compressed BERT model, AutoDistill provides even more compact designs with 3.2% higher accuracy and 1.44× faster hardware performance.

The above scenarios introduce various design spaces for building DNN models and their hardware accelerators to meet real-life requirements. These design spaces contain many different design dimensions and configurable parameters, increasing the difficulty of finding the optimal designs. Since the design spaces are highly application-specific, finding a unified solution to effectively represent them and search for optimal design spots from these spaces is still very challenging.

To improve our proposed designs, we will cover the investigation of unified design space representation and more efficient design space exploration methods. We believe AI hardware acceleration will involve more effective and comprehensive design methods in the future, covering AI algorithms, customized accelerators, and co-design and co-optimization strategies. Our future works will cover more advanced software/hardware co-design for emerging AI models running on heterogeneous systems, which contains a much larger design space and is thus more challenging. Major directions include 1) efficient hardware accelerator designs for handling sparse AI workloads, 2) heterogeneous computing support, which intends to enable system-level design and optimization for heterogeneous AI systems, 3) dynamic computational graph scheduling, which enables the generation of runtime adaptive accelerators for future AI applications, and 4) further investigation of a unified design-space representation and more advanced design-space exploration methods, such as machine-learning based methods.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the Advances in neural information processing systems*, 2012.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton et al., "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[3] OpenAI, "Ai and compute," 2018. [Online]. Available: https://openai.com/blog/ai-and-compute/

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016.

[5] B. Brown, "Intel® math kernel library for deep learning networks," 2018. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-mkl-dnn-part-1-library-overview-and-installation

[6] D. Franklin, "NVIDIA Jetson TX2 delivers twice the intelligence to the edge," *NVIDIA Accelerated Computing— Parallel For all*, 2017.

[7] D. Franklin, "NVIDIA Jetson AGX Xavier delivers 32 teraops for new era of AI in robotics," *NVIDIA Accelerated Computing— Parallel For all*, 2018.

[8] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015.

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

[10] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2018.

[11] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, 2016.

[12] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[13] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers et al., "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[14] K. Wakabayashi, "System LSI design with c-based behavioral synthesis and verification," in *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, 2005.

[15] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the ACM international conference on Multimedia*, 2014.

[16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "Tensorflow: A system for large-scale machine learning," in *Proceedings of the USENIX symposium on operating systems design and implementation (OSDI)*, 2016.

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., "Pytorch: An imperative style, high-performance deep learning library," in *Proceedings of the Advances in neural information processing systems*, 2019.

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

[19] X. Zhang, H. Lu, C. Hao, J. Li, B. Cheng, Y. Li, K. Rupnow, J. Xiong, T. Huang, H. Shi et al., "SkyNet: a hardware-efficient method for object detection and tracking on embedded systems," in *Proceedings of the Conference on Machine Learning and Systems (MLSys)*, 2020.

[20] X. Zhang, D. Wang, P. Chuang, S. Ma, D. Chen, and Y. Li, "F-CAD: A framework to explore hardware accelerators for codec avatar decoding," in *Proceedings of the Design Automation Conference (DAC)*, 2021.

[21] X. Zhang, Z. Zhou, D. Chen, and Y. E. Wang, "AutoDistill: an end-to-end framework to explore and distill hardware-efficient language models," *arXiv preprint arXiv:2201.08539*, 2022.

[22] Y. Li, X. Zhang, and D. Chen, "CSRNet: Dilated convolutional neural networks for understanding the highly congested scenes," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2018.

[23] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2015.

[24] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of Machine Learning Research*, vol. 3, no. 2, pp. 1137–1155, 2003.

[25] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proceedings of the Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the Advances in neural information processing systems*, 2017.

[27] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston et al., "The youtube video recommendation system," in *Proceedings of the ACM Conference on Recommender Systems*, 2010.

[28] S. A. Shaya, N. Matheson, J. A. Singarayar, N. Kollias, and J. A. Bloom, "Intelligent performance-based product recommendation system," 2010, US Patent 7,809,601.

[29] S. Lombardi, J. Saragih, T. Simon, and Y. Sheikh, "Deep appearance models for face rendering," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, pp. 1–13, 2018.

[30] S.-E. Wei, J. Saragih, T. Simon et al., "VR facial animation via multiview image translation," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–16, 2019.

[31] H. Chu, S. Ma, F. D. la Torre, S. Fidler, and Y. Sheikh, "Expressive telepresence via modular codec avatars," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2020, pp. 330–345.

[32] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[33] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[34] A. Graves and N. Jaitly, "Towards end-to-end speech recognition with recurrent neural networks," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2014.

[35] X. Zhang, Y. Ma, J. Xiong, W.-M. W. Hwu, V. Kindratenko, and D. Chen, "Exploring HW/SW co-design for video analysis on CPU-FPGA heterogeneous systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 6, pp. 1606–1619, 2022.

[36] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[37] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[38] D. Gope, G. Dasika, and M. Mattina, "Ternary hybrid neural-tree networks for highly constrained IoT applications," 2019.

[39] Y. Chen, K. Zhang, C. Gong, C. Hao, X. Zhang, T. Li, and D. Chen, "T-DLA: An open-source deep learning accelerator for ternarized dnn models on embedded FPGA," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019, pp. 13–18.

[40] C. Gong, T. Li, Y. Lu, C. Hao, X. Zhang, D. Chen, and Y. Chen, "μl2q: An ultra-low loss quantization method for dnn compression," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, 2019, pp. 1–8.

[41] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proceedings of the Advances in neural information processing systems*, 2015.

[42] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *Proceedings of the Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[43] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of the International Conference on Computer Vision (ICCV)*, 2017.

[44] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2015.

[45] X. Dai, H. Yin, and N. K. Jha, "Nest: A neural network synthesis tool based on a grow-and-prune paradigm," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1487–1497, 2019.

[46] A. Ren, T. Zhang, S. Ye, J. Li, W. Xu, X. Qian, X. Lin, and Y. Wang, "ADMM-NN: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[47] X. Ding, G. Ding, J. Han, and S. Tang, "Auto-balanced filter pruning for efficient convolutional neural networks," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[48] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[49] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," *arXiv preprint arXiv:1711.07128*, 2017.

[50] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proceedings of the International Symposium on Microarchitecture*, 2016.

[51] L. Floridi and M. Chiriatti, "GPT-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, no. 4, pp. 681–694, 2020.

[52] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*. Association for Computational Linguistics, 2019, pp. 4171–4186.

[53] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[54] S. Sun, Y. Cheng, Z. Gan, and J. Liu, "Patient knowledge distillation for BERT model compression," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 4323–4332.

[55] R. Tang, Y. Lu, L. Liu, L. Mou, O. Vechtomova, and J. Lin, "Distilling task-specific knowledge from BERT into simple neural networks," *arXiv preprint arXiv:1903.12136*, 2019.

[56] H. Tsai, J. Riesa, M. Johnson, N. Arivazhagan, X. Li, and A. Archer, "Small and practical BERT models for sequence labeling," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 3632–3636.

[57] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, and D. Zhou, "MobileBERT: a compact task-agnostic BERT for resource-limited devices," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020, pp. 2158–2170.

[58] J. Xu, X. Tan, R. Luo, K. Song, J. Li, T. Qin, and T.-Y. Liu, "NAS-BERT: Task-agnostic and adaptive-size bert compression with neural architecture search," in *Proceedings of ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2021, p. 1933–1943.

[59] J. Zhang and J. Li, "Improving the performance of opencl-based fpga accelerator for convolutional neural network," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[60] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong, "TGPA: tile-grained pipeline architecture for low latency cnn inference," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2018.

[61] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen, "FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge," in *Proceedings of the Design Automation Conference (DAC)*. IEEE, 2019.

[62] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable fpgas," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

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

[64] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *Proceedings of the International Symposium on Microarchitecture*. IEEE, 2020, pp. 766–780.

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

[66] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.-W. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs," in *Proceedings of the Design Automation Conference (DAC)*, 2017.

[67] C. Zhuge, X. Liu, X. Zhang, S. Gummadi, J. Xiong, and D. Chen, "Face recognition with hybrid efficient convolution algorithms on FPGAs," in *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*, 2018.

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

[69] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-DNN: An open framework for mapping dnn models to cloud FPGAs," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 73–82.

[70] Q. Li, X. Zhang, J. Xiong, W.-M. Hwu, and D. Chen, "Efficient methods for mapping neural machine translator on FPGAs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1866–1877, 2020.

[71] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2018.

[72] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "HybridDNN: A framework for high-performance hybrid DNN accelerator design and implementation," in *Proceedings of the Design Automation Conference (DAC)*, 2020.

[73] S. Huang, K. Wu, H. Jeong, C. Wang, D. Chen, and W.-M. Hwu, "PyLog: An algorithm-centric Python-based FPGA programming and synthesis flow," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2015–2028, 2021.

[74] Y. Yang, Q. Huang, B. Wu et al., "Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded FPGAs," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

[75] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of FPGA-based neural network inference accelerators," *ACM Trans. Reconfigurable Technol. and Syst.*, vol. 12, no. 1, pp. 1–26, 2019.

[76] W. Jiang, X. Zhang, E. H.-M. Sha, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, "Accuracy vs. efficiency: Achieving both through FPGA-implementation aware neural architecture search," in *Proceedings of the Design Automation Conference (DAC)*, 2019.

[77] J. Wang, X. Zhang, Y. Li, and Y. Lin, "Exploring HW/SW co-optimizations for accelerating large-scale texture identification on distributed GPUs," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2021, pp. 1–10.

[78] Y. Fu, Y. Zhang, C. Li, Z. Yu, and Y. Lin, "A3C-S: Automated agent accelerator co-search towards efficient deep reinforcement learning," in *Proceedings of the Design Automation Conference (DAC)*, 2021, pp. 13–18.

[79] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 1997–2017, 2019.

[80] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2018.

[81] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2019, pp. 10 734–10 742.

[82] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2019, pp. 2820–2828.

[83] Y. Li, C. Hao, X. Zhang, X. Liu, Y. Chen, J. Xiong, W.-m. Hwu, and D. Chen, "EDD: Efficient differentiable dnn architecture and implementation co-search for embedded AI solutions," *Proceedings of the Design Automation Conference (DAC)*, 2020.

[84] W. Niu, X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, and B. Ren, "Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 907–922.

[85] J. Lin, W.-M. Chen, J. Cohn, C. Gan, and S. Han, "Mcunet: Tiny deep learning on iot devices," in *Proceedings of the Advances in neural information processing systems*, 2020.

[86] A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, and L. Benini, "XpulpNN: accelerating quantized neural networks on risc-v processors through isa extensions," in *Proceedings of the Design, Automation & Test in Euro. Conf. & Exhibition (DATE)*, 2020, pp. 186–191.

[87] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[88] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[89] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2014.

[90] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2017.

[91] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2009.

[92] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2012.

[93] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "Deep learning with int8 optimization on Xilinx devices," *White Paper*, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf

[94] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl deep learning accelerator on arria 10," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[95] H. Zeng, R. Chen, C. Zhang, and V. Prasanna, "A framework for generating high throughput CNN implementations on FPGAs," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018.

[96] Intel, "Enabling high-performance DSP applications with Stratix V variable-precision DSP blocks." [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01131-stxv-dsp-architecture.pdf

[97] X. Xu, X. Zhang, B. Yu, X. S. Hu, C. Rowen, J. Hu, and Y. Shi, "Dac-sdc low power object detection challenge for uav applications," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[98] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2019.

[99] T. Elsken, J. H. Metzen, and F. Hutter, "Efficient multi-objective neural architecture search via lamarckian evolution," in *Proceedings of the Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[100] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2018.

[101] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[102] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016, pp. 779–788.

[103] F. Xiong, S. Yin, Y. Fan, and P. Ouyang, "DAC-SDC'19 2nd place winner in GPU track," 2019.

[104] J. Deng, T. Shen, X. Yan, Y. Chen, H. Zhang, R. Wang, P. Zhou, and C. Zhuo, "DAC-SDC'19 3rd place winner in GPU track," 2019.

[105] H. Lu, X. Cai, X. Zhao, and Y. Wang, "DAC-SDC'18 1st place winner in GPU track," https://github.com/lvhao7896/DAC2018, 2018, accessed: 2020-02-28.

[106] J. Deng and C. a. Zhuo, "DAC-SDC'18 2nd place winner in GPU track," https://github.com/jndeng/DACSDC-DeepZ, 2018, accessed: 2020-02-28.

[107] C. Zang, J. Liu, Y. Hao, S. Li, M. Yu, Y. Zhao, M. Li, P. Xue, X. Qin, L. Ju, X. Li, M. Zhao, and H. Dai, "DAC-SDC'18 3rd place winner in GPU track," https://github.com/xiaoyuuuuu/dac-hdc-2018-object-detection-in-Jetson-TX2, 2018, accessed: 2020-02-28.

[108] B. Zhao, W. Zhao, T. Xia, F. Chen, L. Fan, P. Zong, Y. Wei, Z. Tu, Z. Zhao, Z. Dong, and P. Ren, "DAC-SDC'19 2nd place winner in FPGA track," 2019.

[109] K. Kara and G. Alonso, "DAC-SDC'19 3rd place winner in FPGA track," 2019.

[110] S. Zeng, W. Chen, T. Huang, Y. Lin, W. Meng, Z. Zhu, and Y. Wang, "DAC-SDC'18 1st place winner in FPGA track," https://github.com/hirayaku/DAC2018-TGIIF, 2018, accessed: 2020-02-28.

[111] K. Kara, C. Zhang, and G. Alonso, "DAC-SDC'18 2nd place winner in FPGA track," https://github.com/fpgasystems/spooNN, 2018, accessed: 2020-02-28.

[112] C. Hao, Y. Li, S. H. Huang, X. Zhang, T. Gao, J. Xiong, K. Rupnow, H. Yu, W.-M. Hwu, and D. Chen, "DAC-SDC'18 3rd place winner in FPGA track," https://github.com/onioncc/iSmartDNN, 2018, accessed: 2020-02-28.

[113] L. Huang, X. Zhao, and K. Huang, "Got-10k: A large high-diversity benchmark for generic object tracking in the wild," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 5, pp. 1562–1577, 2019.

[114] B. Li, W. Wu, Q. Wang, F. Zhang, J. Xing, and J. Yan, "Siamrpn++: Evolution of siamese visual tracking with very deep networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2019.

[115] Q. Wang, L. Zhang, L. Bertinetto, W. Hu, and P. H. Torr, "Fast online object tracking and segmentation: A unifying approach," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2019.

[116] N. Xu, L. Yang, Y. Fan, D. Yue, Y. Liang, J. Yang, and T. Huang, "Youtube-vos: A large-scale video object segmentation benchmark," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[117] Qualcomm, "Snapdragon 865 5G mobile platform," accessed: 2022-07-12. [Online]. Available: https://www.qualcomm.com/products/snapdragon-865-5g-mobile-platform

[118] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*, 2018, pp. 2227–2237.

[119] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized autoregressive pretraining for language understanding," *Proceedings of the Advances in neural information processing systems*, pp. 5753–5763, 2019.

[120] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[121] Z. Zhang, X. Han, Z. Liu, X. Jiang, M. Sun, and Q. Liu, "Ernie: Enhanced language representation with informative entities," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019, pp. 1441–1451.

[122] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A lite BERT for self-supervised learning of language representations," in *Proceedings of the Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[123] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2017, pp. 1487–1495.

[124] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *Proceedings of the Advances in neural information processing systems*, vol. 25, 2012.

[125] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[126] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "TinyBERT: Distilling bert for natural language understanding," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020, pp. 4163–4174.

[127] H. Tsai, J. Ooi, C.-S. Ferng, H. W. Chung, and J. Riesa, "Finding fast transformers: One-shot neural architecture search by component composition," *arXiv preprint arXiv:2008.06808*, 2020.

[128] D. Chen, Y. Li, M. Qiu, Z. Wang, B. Li, B. Ding, H. Deng, J. Huang, W. Lin, and J. Zhou, "AdaBERT: Task-adaptive BERT compression with differentiable neural architecture search," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*. Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI), 2020, pp. 2463–2469.

[129] S. Li, M. Tan, R. Pang, A. Li, L. Cheng, Q. V. Le, and N. P. Jouppi, "Searching for fast model families on datacenter accelerators," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2021, pp. 8085–8095.

[130] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "Fitnets: Hints for thin deep nets," *arXiv preprint arXiv:1412.6550*, 2014.

[131] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Proceedings of the Advances in neural information processing systems*, 2011.

[132] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2015.

[133] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *Proceedings of the Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[134] J. Gao, H. Xu, X. Ren, P. L. Yu, X. Liang, X. Jiang, Z. Li et al., "AutoBERT-Zero: Evolving bert backbone from scratch," *arXiv preprint arXiv:2107.07445*, 2021.

[135] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma et al., "Ten lessons from three generations shaped google's TPUv4i: Industrial product," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1–14.

[136] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Large batch optimization for deep learning: Training BERT in 76 minutes," in *Proceedings of the Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[137] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proceedings of the International Conference on Computer Vision (ICCV)*, 2015, pp. 19–27.

[138] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf et al., "Mlperf training benchmark," *arXiv preprint arXiv: 1910.01500*, 2019.

[139] X.-S. Yang, "Firefly algorithm, stochastic test functions and design optimisation," *International journal of bio-inspired computation*, vol. 2, no. 2, pp. 78–84, 2010.

[140] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *Proceedings of the Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[141] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ questions for machine comprehension of text," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2016, pp. 2383–2392.