Fast and Efficient Deep Learning Deployments via Learning-based Methods

SUN, Qi

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
July 2022

Abstract of thesis entitled:

Fast and Efficient Deep Learning Deployments via Learning-based Methods

Submitted by SUN, Qi

for the degree of Doctor of Philosophy

at The Chinese University of Hong Kong in July 2022

The past few years witnessed the significant success of deep learning (DL) algorithms and the increasing deployment efficiency and performance requirements. Features and weights in the deep learning operations usually have prominent scales, and the numbers of computations and consumptions of memory are huge. Various types of operations, *e.g.*, fully connected and convolutional operations, challenge deployment strategies on different platforms, *e.g.*, FPGA and GPU.

Great optimization efforts have been made for a wide range of deep learning algorithms and hardware platforms. Empirics can be summarized to guide the new tasks. We propose to adopt the learning-based methods, that is, learning from known data to drive the deployments of new algorithms. Learning-based techniques can release researchers and designers from the complicated and cumbersome design flow and improve performance automatically and incrementally, with high flexibility compared with the heuristic or analytical methods.

This thesis outlines several methodologies facilitating the deployments of deep learning algorithms in different scenarios. It is mainly composed of the following themes.

- A correlated multi-objective multi-fidelity Gaussian process model with deep kernel functions is proposed to handle the complicated optimization flow of the FPGA-based model deployments. The Bayesian optimization (BO) method is adopted to explore the optimal designs efficiently. Furthermore, the deep neural network is introduced to improve the kernel functions in the Gaussian process model.

- A deep Gaussian transfer learning algorithm is proposed to utilize the historical optimization data to learn the hidden knowledge related to model structures, hardware characteristics of GPU, optimal deployment strategies, *etc.*, and to transfer the knowledge to optimize the new deployment tasks on GPU.

- Further, to preserve the computational graph information and learn more accurate performance models, a graph attention network is designed to extract the structural information via a graph neural network and delve into the complicated relationship between the features via a multi-head self-attention module.

We expect that the thesis does contribute to fast and efficient deep learning deployments, and our ideas will enlighten future studies.

摘要

過往數年見證了深度學習 (DL) 算法的巨大成功以及不斷提高的部署效率和性能要求。在深度學習運算中，特征及權重規模巨大，運算量和內存消耗量也很大。各種類型的運算，例如全連接運算和卷積運算，對不同平臺（例如 FPGA 和 GPU）上的部署策略提出了挑戰。

針對廣泛的深度學習算法和硬件平臺，人們已經進行了大量的優化工作，從中可以總結經驗以指導新任務。我們提出采用基於學習的方法，即從已知數據中學習知識來驅動新算法的部署。與啟發式或分析方法相比，基於學習的技術可以將研究人員和設計人員從復雜繁瑣的設計流程中解放出來，自動、漸進地提高性能，具有很高的靈活性。

本論文概述了以下幾種有助於在不同場景中部署深度學習算法的方法：

- 提出了具有深度核函數的相關多目標多保真高斯過程模型來處理基於 FPGA 的模型部署的復雜優化流程。采用貝葉斯優化 (BO) 方法來有效地探索最優設計。進一步地，引入深度神經網絡來改進高斯過程模型的核函數。

- 提出了一種深度高斯遷移學習算法，利用歷史優化數據學

習模型結構、GPU 的硬件特性、最優部署策略等相關的隱藏知識，並遷移知識以優化 GPU 上的新部署任務。

- 進一步地，為了保留計算圖信息並學習更準確的性能模型，設計了一個圖註意力網絡，通過圖神經網絡提取圖結構信息，通過多頭自註意模塊挖掘特征間的復雜關係。

我們希望該論文有助於快速高效的深度學習算法部署，我們的想法將為未來的研究提供啟發。

Acknowledgement

I would like to express my forever and sincere appreciation to my supervisor, Prof. Bei YU, for his sophisticated guidance, sparkling enlightenment, and sustained support. Without his supervision, patience, and encouragement, completing this thesis and my Ph.D. studies seems impossible.

Taking this opportunity, I also would like to greatly thank my committee members, Professor Qiang XU, Professor Mingchang YANG, and Prof. Jianlei YANG, who have offered useful feedback throughout the process.

It has been a great honor to work with many great collaborators during my Ph.D. studies. Thanks to Dr. Tinghuan CHEN, and Dr. Hao GENG for fascinating collaborations on the topics of HLS optimization, DNN acceleration, and design space exploration. Thanks to Yuzhe MA, Lu ZHANG, Ran CHEN, Zhuolun Leon HE, Wei LI, Chen BAI, Yuxuan ZHAO, Xufeng YAO, Xinyun ZHANG, Yang BAI, and Siting LIU for heaps of constructive discussions and cooperation in plenty of projects. I would like to thank other intelligent and accomplished collaborators, Wenqian ZHAO, Hongduo LIU, Guojin CHEN, and all other CUDA members. I would like to extend my appreciation to the staff in the CSE department for generously helping me.

I owe the deepest gratitude to my parents for their ever-encouraging and -supporting me to pursue what I am interested in and overcome the difficulties in my life.

This work is dedicated to my loving parents and friends.

Thank you.

Contents

Abstract **i**

Acknowledgement **vi**

1 Introduction **1**

1.1 Backgrounds and Motivations 1

1.2 Thesis Structure and Contributions 6

2 Preliminaries **12**

2.1 FPGA and HLS-based Implementation 13

2.2 GPU-based Implementation 18

3 Optimization of HLS Directives **23**

3.1 Introduction 23

3.2 Preliminaries 31

3.2.1 Bayesian Optimization 32

3.2.2 Multi-objective Optimization 34

3.2.3 Multi-fidelity Optimization 36

3.2.4 Gaussian Process Regression 37

3.2.5 Overview of Our Method 39

3.3 HLS Directive Design Space 40
3.3.1 Design Space and Space Pruning 41
3.3.2 Encoding of Directive Configurations . . . 45
3.4 Correlated Multi-objective Multi-fidelity Models and Deep Kernel Functions 46
3.4.1 Non-linear Multi-Fidelity Model 46
3.4.2 Correlated Multi-Objective Model 48
3.4.3 Combined Model 51
3.4.4 Optimization Based on Deep Kernel Functions 52
3.5 The Overall Optimization Flow 56
3.6 Experimental Results 61
3.6.1 Integration of Optimization and Deployment 62
3.6.2 Objective Selection 63
3.6.3 Benchmarks and Methods 64
3.6.4 Experimental Settings 67
3.6.5 Results and Analysis 70
3.6.6 Ablation Studies on Acquisition Function 73
3.6.7 Ablation Studies on Runtime 74
3.7 Summarization 75

4 Deployment via Deep Gaussian Transfer Learning 76
4.1 Introduction 76
4.2 Transfer Learning Based on Deep Gaussian Processes 80

4.2.1 Motivations 80
4.2.2 Our Automatic Optimization Framework . 82
4.2.3 Deep Gaussian Processes with Stochastic Variational Inference 83
4.2.4 Transfer Knowledge to New Tasks 87
4.3 Experiments 90
4.3.1 Experimental Settings 90
4.3.2 Layer Groups and Historical Data 93
4.3.3 Ablation Studies on the Proposed DGP . 94
4.3.4 Ablation Studies on the Transfer Learning 96
4.3.5 Performance of the Whole framework . . . 97
4.4 Conclusion 100

5 Tuning Computations via Graph Attention Network 101

5.1 Introduction 101
5.2 Preliminaries 105
5.2.1 Computational Graphs 105
5.2.2 Graph Neural Networks and Attention Mechanism 107
5.3 Algorithms 109
5.3.1 Overall Flow of GTuner 109
5.3.2 GAT: Graph Neural Network Module . . . 112
5.3.3 GAT: Multi-head Self-attention Module . 115
5.4 Experiments 118
5.4.1 Implementation Details 119

5.4.2 Ablation Studies on GAT Structure 121
5.4.3 Ablation Studies on GPU Measure Trials . 123
5.4.4 Performance of the Whole framework . . . 124
5.5 Conclusion 125

6 Conclusion 127

6.1 Summary 127
6.2 Possible Future Directions 130

Bibliography 133

List of Figures

1.1 The flow of deep learning deployment and our topics. 6

2.1 A typical seven-level for-loop of a direct convolutional operation. 13
2.2 Three types of array partitioning, `BLOCK`, `CYCLIC`, and `COMPLETE`. 15
2.3 “in_channel_loop” is the innermost loop. 16
2.4 “in_width_loop” is the innermost loop. 16
2.5 The fusion of PixelShuffle, ReduceSum and ShiftMeanAdd. 17
2.6 A brief CUDA programming architecture [66], composed of grids, blocks, threads, and some memories. 19
2.7 The computation workloads are partitioned into blocks and then further split to threads and virtual threads. 20

3.1 (a) The FPGA design flow. (b) HLS pseudocodes and directives. 25

3.2 An example of mapping from design space $\mathcal{X}$ to objective space $\mathcal{Y}$, with two directives and two design objectives (*e.g.*, power and delay). We need to learn a black-box function f to bridge $\mathcal{X}$ and $\mathcal{Y}$, so as to simulate the FPGA design tool. . 33
3.3 The correspondence relationships between the design stages in the FPGA design flow and the multiple models and fidelities in the model level. . . . 36
3.4 The brief flow of our method for HLS directives optimization. 40
3.5 An example of design space pruning method. (a) Code with three loops and two arrays. L1 is the outer loop. L2 and L3 are the inner parallel loops. (b) Graphs of array A and B, and the merged graph. 43
3.6 An example of directive encoding. 45
3.7 Normalized delay values of the three fidelities. The X-axis is the index of the design. We assign indices for these designs according to their directive values in increasing order. (a) `GEMM` (general matrix multiplication). (b) `SPMV_ELLPACK` (sparse matrix-vector multiplication using the ELLPACK format). 47
3.8 An example of minimizing power and delay. . . . 50
3.9 The combined models, with three stages (fidelities) and three objectives. 53

3.10 Our method combines the GP model GP_Θ and the deep kernel function ϕ_W. 55
3.11 Detailed overall optimization flow. 56
3.12 A toy example to explain the models of the 3 stages (as shown in Fig. 3.3) and their corresponding acquisition functions. 60
3.13 Sizes of The Design Spaces Before and After Pruning. 65
3.14 The training loss of our method, and the mean parameters (W and Θ) of the deep kernel functions and the GP models. The results show the great convergence of the training loss. The mean parameters W and Θ follow the same convergence trend which validates the effectiveness of our training method. 69
3.15 Learned Pareto designs of `GEMM` and `SPMV_ELLPACK` in the objective spaces. 70

4.1 Our automatic optimization framework, consists of three stages, *i.e.*, stage 1: DGP model preparation based on the history data, stage 2: transfer knowledge to new DNN layers, and stage 3: optimal configuration searching. 84

4.2 RMSE of our predicted GFLOPS, the data are expressed as the ratios to the results of XGBoost in AutoTVM. Here, our DGP is directly used to predict the GFLOPS of new tasks without tuning. cv: convolution, rb: residual block, sc: shortcut, sp: separable convolution, dp: depthwise convolution. 91
4.3 The randomly sampled tuning set and the set selected according to DGP. The data are in descending order. There are 300 configurations in each tuning set, and the X-axis is the index of the configuration. 95
4.4 Comparisons between AutoTVM and ours. "Selected" means the tuning configurations are selected by using our pre-trained DGP as the criterion. "Random" means the tuning configurations are randomly sampled from the configuration space without any prior knowledge. 96
4.5 The ratios of the GFLOPS values of VGG-16. . . 99

5.1 An example of the sketch. 107
5.2 Two annotations of the code sketch. 108
5.3 The overall flow of our GTuner. 109
5.4 Structure of our graph attention network (GAT). 112

5.5 Typical diverse structures in VGG, ResNet (residual block) and SqueezeNet (fire block). Conv-x denotes that the kernel size is x. 113
5.6 Graph neural network. 115
5.7 Multi-head self-attention with $\boldsymbol{x}$ as the query ($\boldsymbol{Q}$), key ($\boldsymbol{K}$), and value ($\boldsymbol{V}$) simultaneously. 117
5.8 Results of different measure trials. 123
5.9 Detail results of subgraphs in ResNet-18. 124

List of Tables

3.1 Average Runtime of The FPGA Tool for Each Design 62
3.2 Normalized Experimental Results 71
3.3 Profiling Information of Runtime Details (Optimization Costs, hours) 72
3.4 Profiling Information of Overall Runtime (hours) 72
3.5 Comparisons of Normalized ADRS with MES . . 74

4.1 Comparisons of Search Time and End-to-end Model Inference Latency 97

5.1 Comparisons between Convolutional Layers . . . 122
5.2 Performance without GNN or MHSA 122
5.3 End-to-end Model Inference Latency (ms) 125
5.4 Time Costs (minutes) of the Optimization Processes 126

Chapter 1

Introduction

1.1 Backgrounds and Motivations

Deep learning algorithms, *aka.*, deep neural networks (DNNs), have achieved significant developments and shown great success in wide application scenarios, such as object detection [99], natural language processing [35], adversarial defense [118], text recognition [148], physical design [45], and super-resolution algorithms [129]. An interesting trend is that researchers proposed to use larger and deeper models which contain more weights and complicated computation patterns to delve into the tasks, learn more information, and improve the model generalization. However, the enormous computational intensities and heavy communication workload result in sustainable challenges to the fast and efficient model inferences.

In recent years, great efforts have been made to accelerate the inference from various algorithmic perspectives, including quantization [8, 36, 41] to adjust the data precisions, pruning

[22, 81, 120, 156] to remove redundant data, neural architecture search [47, 76, 143] to explore the model structures, hardware-model co-design or hardware-aware neural architecture search [18, 32, 52, 73] to find the inference-efficient model structures or design model-efficient hardware jointly, *etc.*

There are also many studies on designing and optimizing DNN applications on various hardware platforms. [128] uses the systolic array to accelerate convolutional layers on FPGA. [55] implements FSRCNN-based super-resolution systems on FPGA. [80] accelerates the kernel sharing Winograd systolic array on FPGAs though the Winograd operation has limited applications due to the computational constraints on the kernels. [116] fuses operations in DNN models and optimizes the power consumptions on FPGA. [152] optimizes a dictionary learning-based super-resolution model using NVIDIA TensorRT [1] and CUDA programming on GPU. [68] introduces the CUDA graph, multi-stream computing, and bipartite graph match to implement the multi-task parallelism for DNN models on NVIDIA GPU. [144] improves the depthwise and pointwise convolutional layers on a quad-core ARM CPU. [95] quantizes and maps convolutional networks on resistive random access memory (RRAM) accelerators. Devices vendors also provide highly optimized tools by exploiting the details of the low-level hardware implementations, *e.g.*, Intel oneDNN [2], NVIDIA TensorRT [1], and Xilinx Deep Processing Unit (DPU) [5]. CPU provides limited supports for the intensive computation and communication parallelism in

DNN tasks. In contrast, FPGA and GPU have been the most popular platforms thanks to the flexibility and configurability of their task implementations.

Furthermore, researchers propose some techniques to stimulate the fast and efficient deployments of DNN algorithms, due to the complicated model structures and hardware architecture, incomprehensible, time-consuming and expensive mapping process, poor performance estimators, *etc.* Some approaches have been proposed to counteract these challenges. For FPGA-based implementations, some researchers defines some analytical models to measure the performance [21, 40, 52, 73, 116, 139], which usually suffer from a lack of flexibility and accuracy [75, 104]. Researchers develop resource utilization and latency models in these analytical methods, such as utilizations for DSP, BRAM, and LUT, and latencies for computations and communications. The whole performance models are usually built by dividing problems into small modules, calculating performances for these modules, and summarizing these modules finally. Therefore, the details of hardware architectures and model structures are crucial. Similarly, [71] defines analytical models for convolutional neural networks (CNNs) on CPU. In contrast, the complicated patterns and obscure mechanisms for mapping DNN models on GPU hinder utilizing analytical models for GPU-based deployments. Consequently, researchers propose some learning-based approaches which use the machine learning models to map the DNN tasks and hardware platforms to the performance values. AutoTVM [28] introduces the XGBoost to regress the inference latency according to the code embeddings. Based on AutoTVM, an advanced active learning method [114] is designed to learn representative parameters in the optimization process, with the help of transductive experimental design-based initialization, boosting method, and adaptive optimization. CHAMELEON [7] introduces reinforcement learning to learn the searching strategies from the history tuning data and adapts the searching space during optimization. Though reinforcement learning shows promising results in many applications, the results show that performance improvements mainly results from the adaptive searching. Guided Genetic Algorithm (GGA) [89] utilizes some heuristic rules to guide the genetic algorithm. The similarities between new deployment tasks and the historical tuning data are calculated to measure the performance of new designs.

Despite the advancements, existing deployment techniques are still unsatisfying for several reasons. Firstly, the complicated relationships between the input features, performance metrics, and the design stages are not well modeled and clarified. There are three design stages in the FPGA-based design flow (*e.g.*, Vivado flow), *i.e.*, high-level synthesis (HLS), logic synthesis, and implementation. There are three performance metrics (objectives) in each stage, *i.e.*, power, latency, and resource consumption. In other words, each stage reports these three performance values but with different fidelities. The later stages have more information and consequently report more accurate data. The correlated and non-linear relationships between these performance metrics and these three stages are not characterized. Secondly, the traditional optimization process is slow, resulting from the immense design space and time-consuming design flow. The performance models are of low quality, thus failing to find optimum designs efficiently and wasting the optimization overheads. Usually, there are more than millions of designs in the design space. For FPGA, each design consumes hours of synthesis costs. For GPU, compiling the codes and executing the kernels consume several seconds. The learning-based models require collecting the historical optimization data as the training set. The expensive optimization process hinders the applications of the existing methods while these methods fail to measure the performance accurately, thus leading to more optimization steps. Thirdly, the critical information of the historical tuning data is not fully utilized, and some features are ignored despite their physical meanings and relationships. Existing techniques rely on the statistical features while the structural characteristics of the DNN models are discarded. The statistical features are also treated equally, and the implicit importance of information is not taken into consideration.

These drawbacks stimulate our explorations for novel techniques to improve the deployment approaches via learning-based methods, targeting delving into the features, accurately modeling the data, reducing optimization costs, and finding better deployment solutions faster and more efficiently.

1.2 Thesis Structure and Contributions

This thesis attempts to investigate several methodologies lightening the aforementioned challenges in the deployments of deep learning algorithms. The flow of the deployment of deep learning algorithms and the focuses of our methods are shown in Fig. 1.1.

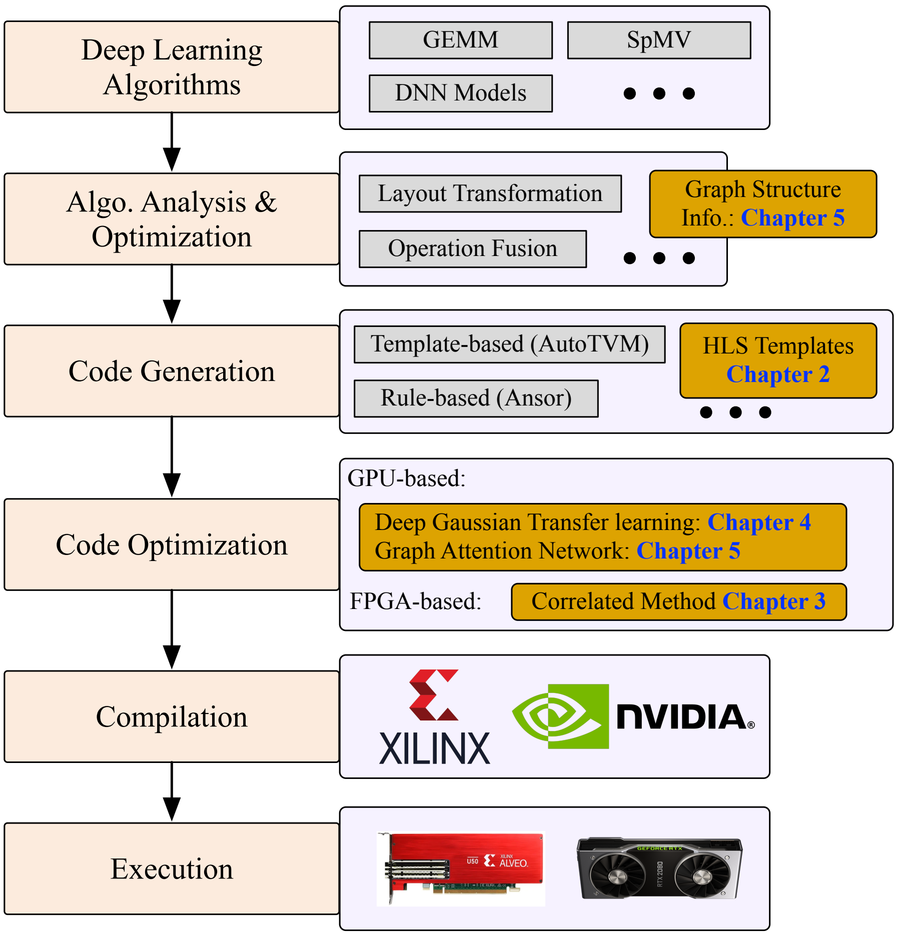


Figure 1.1 The flow of deep learning deployment and our topics.

The first contribution of this thesis is improving the learning model in the HLS directives optimization for the FPGA-based deployments [115]. Bayesian optimization is combined with the multi-fidelity and multi-objective optimization as the optimization framework, to trade-off the model accuracies and optimization workloads of the FPGA design flow. Non-linear multi-fidelity models are built to measure the non-linear relationship between the reports of the three design stages and HLS directives. Correlated multi-objective Gaussian process models are proposed, to tackle both of the correlated relationships among various design objectives and the implicit and complicated mapping relationships between directives and objective values, to find Pareto configurations accurately. Our method is further enhanced to be a deep version (DCGP) by using deep neural networks as the kernel functions, to learn better feature representations flexibly. Therefore the performance of our method can be further improved significantly. Three design objectives, power, delay, and resource consumption are considered as the objectives, thus making the task practical while challenging. We conduct experiments on some public FPGA design benchmarks (including general matrix multiplication and sparse matrix-vector multiplication) and DNN-based object detection model iSmart2. The experimental results show the outstanding performance of our algorithms on the DNN applications and the related applications. The found Pareto designs cover the optimal designs for various design objectives, *i.e.*, power, delay, and resource consumption, and strike a good balance between these objectives. This contribution is detailed in Chapter 3.

The second contribution of this thesis is proposing a novel automatic optimization framework based on deep Gaussian transfer learning, to utilize the historical data to help tune the deep learning deployments on GPU [113]. Firstly, a deep Gaussian process (DGP) model is built on the historical optimization data to learn the hidden knowledge related to model structures, hardware characteristics, optimal deployment strategies, and etc. Stochastic variational inference is adopted to optimize the DGP. Secondly, when deploying a new DNN model, some efficient initial configurations of this new model are sampled under the guidance of the prior knowledge in the pre-trained DGP model. Maximum-a-posteriori (MAP) estimation is applied to tune the DGP model according to these initial configurations, to make the DGP model accommodate for the new task with no loss of the hidden knowledge. Finally, the tuned DGP model is used as a replacement to the time-consuming compilations and on-board inferences during optimization, to predict the performance values of new configurations accurately. Our tuned DGP model accelerates the optimization process remarkably while reducing the inference latency of the final model deployment simultaneously. The representative DNN layers widely used in both industries and academia are tested in these models, including convolutional layers, residual blocks, depthwise separable convolutional layers, and *etc.* The results show that our method outperforms the state-of-the-art baselines significantly.

This contribution is detailed in Chapter 4.

The third contribution of this thesis is delving into the structural features of the deep learning models and using the multi-head self-attention module to learning more information to improve the performance estimator [119]. The structural information of the computational graphs and statistical code features are utilized. The complicated relationships between the features are learned automatically. A novel method, GTuner, is proposed with a graph attention network (GAT) as the performance estimator. GAT comprises a graph neural network (GNN) module to aggregate structural information and a multi-head self-attention (MHSA) module to mine inter-feature relationships. Structural information of the computational subgraphs is extracted from the intermediate representations of the compilation flow with the help of our code parser and analyzer, *i.e.*, in the algorithm analysis and optimization step in Fig. 1.1. Then the information is propagated and aggregated via the graph neural layers to learn high-quality features for the graphs. The MHSA module is designed to learn the complicated but implicit relationships between the structural and code statistical features via the self-attention mechanism. The drawbacks of losing structural information and long-range dependencies between the features are overcome. With the GAT, GTuner optimizes the kernel codes for GPU efficiently. The experimental results demonstrate the remarkable performance of GTuner compared with the baselines. This contribution is detailed in Chapter 5.

Some critical challenges in the deployment process are handled, including the extremely-large design space, time-consuming design flow, task and model transferability, and low-quality models. Our methods can adaptively handle the optimization flow to find good results or stop the expensive flow at the early stages if the results are promising. Further, with our high-quality and high-transferability models, we can find more promising designs in large design spaces with fewer optimization steps. Therefore, our methods can accelerate the optimization processes significantly. For example, in the multi-fidelity multi-objective optimization for the HLS directives, our approach accelerates the optimizations remarkably since we find better designs in each Bayesian optimization step according to our advanced acquisition functions. Besides, in each Bayesian step, we can control the FPGA design flow to stop at early stages (*e.g.*, at the logic synthesis stage) if the results are good enough. The Bayesian optimization also adopts the early stopping techniques to reduce the computation and synthesis costs. In the GPU-based methods, we utilize the simulated annealing with early-stopping techniques and better performance models (*e.g.*, deep Gaussian process model or the multi-head self-attention model). Better results are sampled according to the performance models, and the process stops at good results intelligently.

The structure of the thesis is organized as follows. Chapter 2 provides related backgrounds about some common features designed in different ML-aided design flow stages. Chapter 3 covers the first contribution with corresponding technique details, while the second contribution is introduced in Chapter 4. The third contribution is discussed in Chapter 5. Chapter 6 summarizes this thesis and delivers the possible future study directions.

□ **End of chapter.**

Chapter 2

Preliminaries

For-loops are the mainstreaming operations considered in the deep learning algorithms. Typically, convolutional operations can be represented as a seven-level for-loop, as shown in Fig. 2.1. B is the batch size. M and N are the number of output channels and input channels. H is the height of features, and W is the width of features. KH and KW are the height and width of kernels. The size of weight tensor is $[M, N, KH, KW]$ and the size of input tensor is $[B, N, H, W]$. It is important to organize the hardware resources to conduct communications and computations and schedule these loops, *i.e.*, to determine an optimal deployment configuration. There are some similar operations, such as general matrix multiplication (GEMM) for fully-connected layers, sparse matrix-vector multiplication (SPMV) for sparse computations, *etc.*

Some operations are more complicated, such as softmax, PixelShuffle, ShiftMeanAdd, *etc.*, which are handled by templates or code generation rules. Designers usually suffer from complicated development processes for optimizing the deployment configurations. Urgent requirements are raised for the optimization techniques to ease designers from cumbersome developments.

```
for b in range(0, B):
  for o in range(0, M):
    for i in range(0, N):
      for h in range(0, H):
        for w in range(0, W):
          for kh in range(0, KH):
            for kw in range(0, KW):
              Out[b][o][h][w] += W[o][i][kh][kw]
                    × In[b][i][h+kh][w+kw]
```

Figure 2.1 A typical seven-level for-loop of a direct convolutional operation.

2.1 FPGA and HLS-based Implementation

HLS-based implementations are widely used in academic research for deep learning deployments. In this thesis, the HLS-based implementations are adapted for FPGA deployments and some HLS code templates are designed. Typical optimization techniques include pipelining, unrolling, reshaping, reordering, fusion, *etc.* The HLS directive configurations are termed *knobs*.

Pipelining reduces the initiation interval for a function or loop by allowing the concurrent execution of operations, *e.g.*, read, calculation, and write. This is influenced by the design of computation engines, data lengths, time slots for each operation, *etc.* Unrolling technique flattens the loops to parallelize the computations and communications, thus reducing the system latency at the cost of more resource consumption. Reshaping techniques reorganize the memory blocks to facilitate memory accesses. Typically, the continuous memory allocated to arrays is split into some fragments in the BLOCK, CYCLIC, or COMPLETE manner [4] Fig. 2.2 is taken as an example to illustrate the reshaping techniques. The original array is stored in a continuous memory space. In this example, the partitioning factor is 2. Both the `CYCLIC` and `BLOCK` will partition the array into two separate memory spaces. `CYCLIC` creates smaller arrays by interleaving elements from the original array. `BLOCK` creates smaller arrays from consecutive blocks of the original array. `COMPLETE` decomposes the array into individual elements. For a one-dimensional array, `COMPLETE` corresponds to resolving a memory into individual registers. Usually, reshaping should be compatible with unrolling, which means the data to be computed simultaneously ought to be allocated to different onboard memory blocks to permit parallel fetch. Reordering organizes the loops in different orders for a given task and stimulates different communication and computation patterns, *i.e.*, finishing a task in different inner-task orders. The data accessed in the outer loop are reused by the inner loops, *aka.* data reuse [29]. In this thesis, our HLS code templates have two types of data reuse patterns are provided, input reuse and output reuse, *i.e.*, input-stationary and output-stationary. Some examples of the

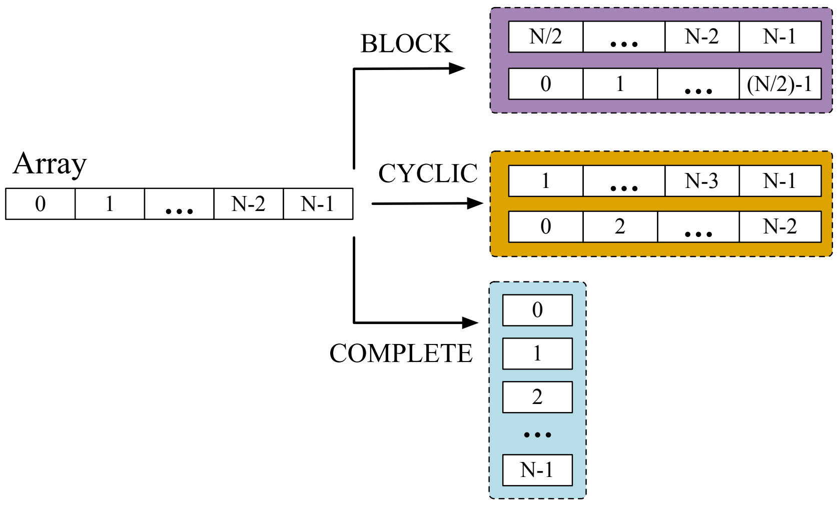


Figure 2.2 Three types of array partitioning, `BLOCK`, `CYCLIC`, and `COMPLETE`.

knobs are shown in Fig. 2.3, including array_partition, unroll, and pipeline with initiation interval (II).

These optimization techniques should be considered elaborately. Optimizing a single technique would possibly have no effect because of the unsuitable usage of other techniques. Some pseudo-code examples of the HLS codes and the HLS knobs are shown in Fig. 2.3 and Fig. 2.4. In Fig. 2.3, "in_channel_loop" is the innermost loop, and unrolling it means the memory should be split cyclically to make both techniques efficient. Executing in_channel calculations in parallel requires 2×in_channel reads and in_channel writes simultaneously. In Fig. 2.4, the innermost loop is "in_width_loop". Unrolling it requires the BLOCK memory partition. The readers can refer to [3] for more details. Each multiplication operation consumes a DSP and three memory communications, two for read and one for write. High parallelism consumes more resources, makes the placement and

```
#pragma HLS ARRAY_PARTITION variable=results dim=0 cyclic factor=in_channel
#pragma HLS ARRAY_PARTITION variable=input_1 dim=0 cyclic factor=in_channel
#pragma HLS ARRAY_PARTITION variable=input_2 dim=0 cyclic factor=in_channel
in_height_loop:
for(int h = 0; h < in_height; h++) {
in_width_loop:
    for(int w = 0; w < in_width; w++) {
in_channel_loop:
#pragma HLS PIPELINE II=1
        for(int c = 0; c < in_channel; c++)
        {
#pragma HLS UNROLL factor=in_channel
            results[h * in_width * in_channel + w * in_channel + c]
                = input_1[h * in_width * in_channel + w * in_channel + c]
                * input_2[h * in_width * in_channel + w * in_channel + c];
        }
    }
}
```

Figure 2.3 “in_channel_loop” is the innermost loop.

```
#pragma HLS ARRAY_PARTITION variable=results dim=0 block factor=in_channel
#pragma HLS ARRAY_PARTITION variable=input_1 dim=0 block factor=in_channel
#pragma HLS ARRAY_PARTITION variable=input_2 dim=0 block factor=in_channel
in_height_loop:
for(int h = 0; h < in_height; h++) {
in_channel_loop:
    for(int c = 0; c < in_channel; c++) {
in_width_loop:
        for(int w = 0; w < in_width; w++)
        {
            results[h * in_width * in_channel + w * in_channel + c]
                = input_1[h * in_width * in_channel + w * in_channel + c]
                * input_2[h * in_width * in_channel + w * in_channel + c];
        }
    }
}
```

Figure 2.4 “in_width_loop” is the innermost loop.

routing congested, thus resulting in timing challenges.

A more complicated example is the fusion of PixelShuffle, ReduceSum, and ShiftMeanAdd from a super-resolution model WDSR [142]. PixelShuffle is to up-sample/reshape the input features by merging the features from several channels into a single channel (*i.e.*, compressing the number of channels, and expanding the features in the result channel) [107]. ReduceSum operations (*aka.*, reduction) sum values from all of the channels. In Fig. 2.5, inputs of this example have 4 channels. The upscale factor of PixelShuffle is 2. Therefore the output has 1

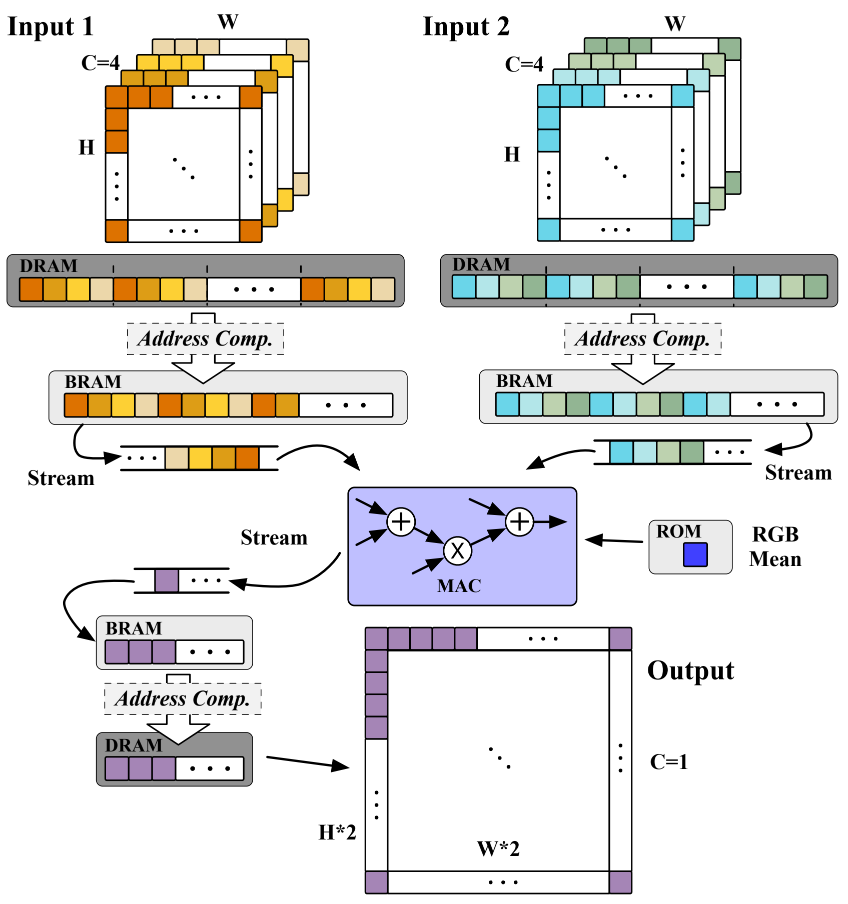


Figure 2.5 The fusion of PixelShuffle, ReduceSum and ShiftMeanAdd.

channel and the height and width are 2× of the inputs. The PixelShuffle is implemented via reading and writing data at specific addresses. The ReduceSum and ShiftMeanAdd are implemented in the MAC:

$$out = (in_1 + in_2) * 127.5 + rgb\_mean[oc], \qquad (2.1)$$

where 127.5 is a constant scale value and *oc* denotes the output channel. The constant *rgb_mean* (RGB Mean) values of the ShiftMeanAdd are stored in the onboard ROM. The HLS dataflow stream technique is also adopted. Each data item read into the stream queue will be pumped into the MAC, do the calculations, and be written out to the output stream. This process executes sequentially, continuously, and steadily without blocking. Therefore, one MAC is adequate while the computation speed is fully promoted.

Optimal performance attributes not only to the optimized HLS code but also the knobs, *e.g.*, the factors controlling the parallelism and memory sizes. The onboard resources also constrain the algorithm deployments. For example, there are 504K system logic cells, 38Mb memory, 461K CLB flip-flops, 1728 DSPs, 1 Quad core Arm Cortex-A53 MPCore, and 1 Dual core Arm Cortex-R5 MPCore. The complicated design mechanism makes the optimization problem non-trivial. Suitable learning methods are essential which are the key topics of this thesis.

2.2 GPU-based Implementation

The NVIDIA CUDA [66] is taken as an example to explain the programming abstraction architecture on GPU, as shown in Fig. 2.6. The programming architecture is composed of grids, blocks, and threads, and some memories. It provides fine-grained data parallelism, thread parallelism, nested within coarse-grained data parallelism, and task parallelism. The dense computational task is partitioned into smaller sub-tasks that can be conducted independently in parallel in these blocks. Following the single instruction multiple threads (SIMT) mechanism, each block is partitioned into a group of threads that can run the same code on different data synchronously. To further improve the performance, the virtual threads (VT) technique is developed to utilize the resources efficiently, ignore the schedule limits, avoid resource conflicts, and reduce the logic complexity of management [141]. Threads are categorized into active and inactive groups according to their states, *e.g.*, computing or blocking. GPU scheduler switches between active and inactive threads to keep the hardware in use without a long system stall.

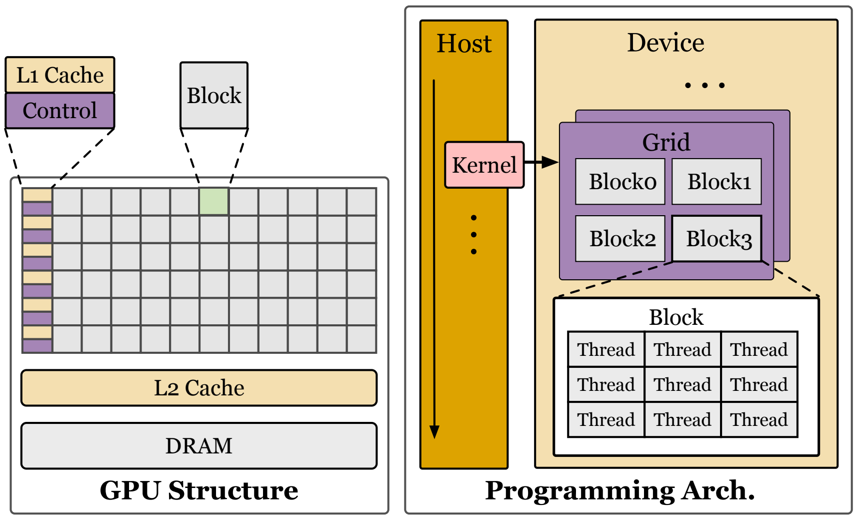


Figure 2.6 A brief CUDA programming architecture [66], composed of grids, blocks, threads, and some memories.

An important characteristic of deploying DNN models on these platforms is to maximize parallelism. To schedule DNN operations on GPU, the computation workloads are assigned to these grids, blocks, virtual threads, and threads. Fig. 2.7 is

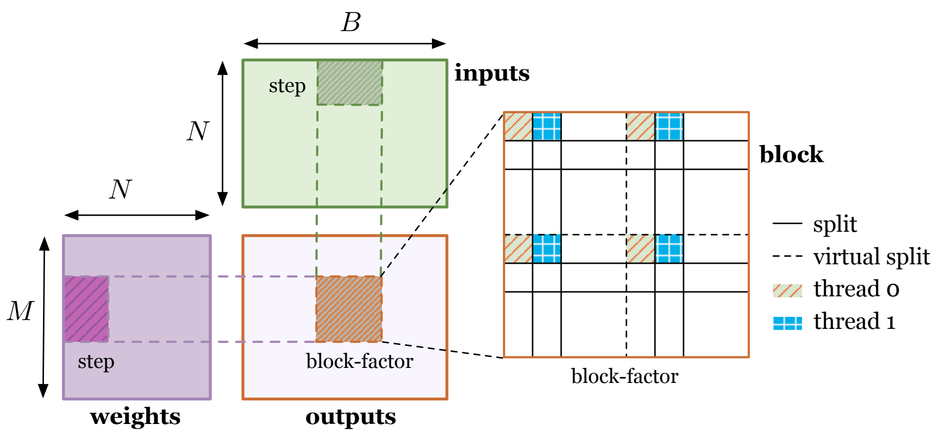


Figure 2.7 The computation workloads are partitioned into blocks and then further split to threads and virtual threads.

taken as an example to illustrate how to partition the workloads of DNN models and map them to hardware. For simplicity, the input tensor and weight tensor are represented as matrices with sizes $N \times B$ and $M \times N$, respectively. Firstly, the input and weight are split into small rectangles, with sizes *step* $\times$ *block-factor* and *block-factor* $\times$ *step*. The size of the corresponding outputs is *block-factor* $\times$ *block-factor*. To get the result of each output rectangle, its corresponding input and weight rectangles are assigned to a CUDA block to conduct the computations. Secondly, the computations are further split into *block-factor* $\times$ *block-factor* threads. Then these threads are assigned into some virtual groups to be scheduled by the CUDA runtime system.

To determine the optimal deployment configuration, some automatic flows are developed, among which TVM [26] is widely used. In TVM, deployment configurations of layers in a DNN model are optimized layer by layer. In Fig. 2.1, each of the for-loops on M, H, and W is split into four sub-loops. These four sub-loops are mapped to blocks, virtual threads, threads, and in-thread-for-loops, respectively. The bound of each sub-loop reflects the number of the allocated hardware resources. Each of the for-loops on N, KH, and KW is split into two sub-loops. These two sub-loops are mapped to threads, and in-thread-for-loops, respectively. The detailed information of deployment configurations is in the appendix. To determine the number of resources allocated to these sub-loops, *i.e.*, the bounds of these sub-loops, a comprehensive search space is defined, in which all of the possible configurations to the resource allocations are contained. The search space is usually composed of millions of configurations.

The limited GPU resources concurrently restricts the computation patterns with respect to the threads, blocks, *etc.* Different GPUs have distinct compute capabilities. For clarity, the edge embedded GPU NVIDIA Jetson Xavier NX which uses the Volta microarchitecture is taken as an example to illustrate this. From the perspective of hardware architectures, there are 6 streaming multiprocessors (SMs) in it. Each SM occupies a 96 KB shared memory/L1 cache. Each SM is further partitioned into 4 processing blocks. Every processing block has 16 FP32 cores, 8 FP64 cores, 16 INT32 cores, 2 Tensor cores, and a 64 KB shared register file. Besides, each processing block has a warp scheduler, to schedule the threads assigned to this processing block. From the perspective of the programming model, the computation kernel is executed as a grid of thread blocks. Each thread block (different from the processing block mentioned above) is assigned to a single streaming multiprocessor. Once the block is scheduled to an SM, threads in this block are further partitioned into warps. Every warp consists of 32 consecutive threads and all threads in a warp are executed in SIMT fashion. While the warps within a thread block may be scheduled in any order, the number of active warps is limited by SM resources. Four processing blocks in every SM of NX means there are at most four active warps in executing at the same moment. Besides, the number of warps in a thread block is constrained by the programming model to fit the sizes of warp schedulers, instruction registers, and *etc.* Sharing data in the shared register files among the parallel threads in the same processing block, or sharing data among the processing blocks in the same SM may cause a race condition: multiple threads accessing the same data in the memory simultaneously. Once a warp idles for the race conditions, the SM is free to schedule other available warps.

For clearness, all of the deployment settings (*e.g.*, bindings of blocks, and threads) to be determined are encoded as the attributes of a feature vector which is termed as *deployment configuration*. A deployment configuration can be denoted as a feature vector $\boldsymbol{x}$. This is the similar to *knob* mentioned above. For clarity, we do not distinguish between these two concepts.

□ **End of chapter.**

Chapter 3

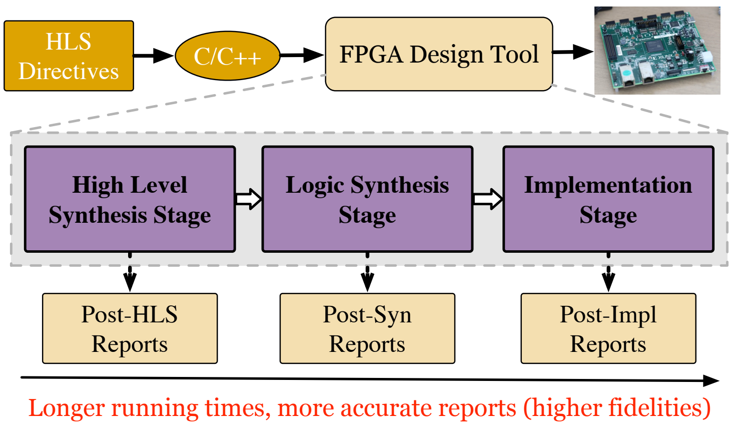
Optimization of HLS Directives

3.1 Introduction

The FPGA design flow is complicated, typically composed of several different steps or phases, including design entry, logic synthesis, and implementation (*aka.*, placement-and-routing). Design entry is to describe the functionalities by the hardware description languages (HDLs). Logic synthesis turns the HDLs into a design implementation in terms of logic gates. Implementation conducts the placement and routing and generates the bitstream. The workload of the whole flow is heavy and time-consuming. Further, high-level synthesis (HLS) tools, used as the design entry tools, have made it possible for users who are not experts in writing HDLs to describe their FPGA designs, by translating high-level programming languages (*e.g.*, C/C++) to low-level HDLs, under the guidance of HLS directives. An example of the Xilinx FPGA design flow relying on HLS is illustrated in Fig. 3.1(a). C/C++ source code and HLS directives are fed into the design tool. There are three analysis stages and the later stages obtain more accurate reports but consume longer running times. The HLS directives are embedded into C/C++ source code, as the inputs to the FPGA design tool. Fig. 3.1(b) shows the pseudo-codes and directives. The directives are in the boxes, beginning with "`#PRAGMA`". Each directive has some factors, *e.g.*, ON and OFF of INLINE, and 2, 5, and 10 of UNROLL. Briefly, in this work, our task is to determine the optimal factors for each of these directives.

HLS directives guide the translation process of the high-level language descriptions, in terms of how to parallelize the computations, how to allocate the memory and computation resources, and *etc.* Given different HLS directive configurations, the final hardware architectures generated from the same high-level language description may vary a lot from each other and therefore have distinctive performance values. In the problem of HLS directives design, the target is to find some HLS directives designs that optimize the design objectives from the entire design space which is composed of candidate directives designs. Some common performance objectives include power, delay, and resource consumption. We need to choose the best factor for each directive to obtain the best performance values. With these advantages, HLS tools have been widely used in many applications, *e.g.*, floating-point computations [12, 74], and deep neural network deployments [52, 127].

Several problems still hinder researchers from finding the op-



(a)

```
comp(int in[10], int out[10]):
  #PRAGMA HLS INLINE={ON, OFF}
  for(i = 0; i < 10; i ++) {
    #PRAGMA HLS UNROLL factor={2,5,10}
      in[i] = out[i];
  }
```

(b)

Figure 3.1 (a) The FPGA design flow. (b) HLS pseudo-codes and directives.

timal directives design efficiently. Firstly, it is difficult to find designs that balance the multiple design objectives. For example, reducing system delay demands higher parallelisms which would require more computation cores and have higher resource consumptions, and vice versa. Therefore, optimizing the multiple objectives simultaneously is a multi-objective optimization problem. Secondly, the whole design flow is time-consuming and the analysis reports of these several stages have different fidelities. Later stages can report more accurate analyses, at the cost of longer running times. This kind of multi-stage design problem is also called multi-fidelity design. Besides, the reported results of the three stages in Fig. 3.1(a) and the HLS directives are usually in complicated relationships which make it difficult to map between them. We cannot guarantee whether a design is good or valid in the Implementation stage, though its HLS estimated performance is good. Therefore, we need to predict the quality of the reports at each stage to determine whether we need to run the later FPGA design stages to get more accurate reports.

Some efforts have been made to facilitate the selection of HLS directives. Several analytical/synthesis methods were proposed to analyze the HLS directives, to estimate the performance with no need of running the FPGA design flow for too many directives designs. For the general applications, the directive configurations are analyzed by using an analytic model or a simulator, *e.g.*, Lin-analyzer [155], COMBA [149, 150], polyhedral model [158] and *etc.* Some proposed to use the dedicated heuristics methods, *e.g.*, lattice [42], clustering [102], divide and conquer [103], and greedy method [94] to guide the exploration of the directive designs. For the deep neural network applications, researchers proposed complicated formulations to simulate the systolic arrays for convolutional operations [116, 128], Spatial/Winograd convolution [139], or several specific code templates [52]. However, these works depend on the accuracy of the analytical models and lack generality or still consume much time to conduct the detailed code analysis and synthesis (*aka.*, profiling) to determine the parameters in the analytical models [52,116,128,139]. Typical parameters include the unit costs of latency, power, and resource consumptions for the given application and FPGA device, which vary significantly under different scenarios. For new applications, new analytic formulations are required, especially for the complicated deep neural networks. These applications challenge the generality of these analytical methods. Besides, the feature of the multiple stages (multi-fidelity) in the FPGA design flow is not considered in these works.

Some model-based works use machine learning algorithms to map from the HLS directives to the performance values, where the complicated FPGA design stages are regarded as black-box functions which can be modeled by machine learning algorithms. Compared to the analytical methods, model-based methods are more flexible and general. The inputs to these models are the feature encodings of the HLS directives and the predicted outputs are the performance values. In these works, the authors collect lots of data to train machine learning models. [86] uses simulated annealing to collect training data, to train a decision tree to guide the exploration of new designs. [77] uses randomized transductive experimental design (RTED) to draw efficient designs from the design space. [78] guides the FPGA designs according to the known ASIC designs by training a regressor, with the help of [77] as the initialization method. Similarly, [27,33,79,91,121,151] propose to use more machine learning algorithms to predict the routing congestions, power, performance and *etc.*, according to the reports and results at various design stages, or the reports of some preliminary analytic models. Some typical algorithms include linear regression, artificial neural networks, and boosting trees. However, huge amounts of real design reports are necessary to guarantee accuracy due to the limited performance of these models. The multiple objectives are usually considered independently, and a machine learning model is built for each objective separately. Besides, the multi-fidelity reports are also not utilized. They use the *post-Implementation* reports at the cost of longer running times or use the *post-HLS* reports at the cost of data accuracies without considering the trade-offs are between running times and data accuracies. To reduce the simulation costs and make full use of the existing reports, recently, Bayesian optimization (BO) approaches based on the Gaussian process (GP) have been proposed. However, the multiple objectives are independent of each other [82]. This work is further extended by considering linear multi-fidelity designs [83]. Wider communities have discussed similar tasks, *e.g.*, high-speed adder [44].

However, it is regrettable that some important characteristics of the directive design are ignored. Firstly, the multiple design objectives are in complex correlated relationships. The correlated relationship has been proven to be an important factor in various applications in practical scenarios [24, 25]. Therefore, there exist losses of accuracy in the previous works since they build some independent models to characterize the design objectives. Secondly, the performance values of the three design stages and the directives are in non-linear relationships. It is hard for the designers who implement the high-level descriptions to estimate the performance of the design after placement and routing. Some ignore or evade these complicated relationships by only considering the lowest fidelity (*post-HLS* reports), though they miss some data from the later stages. Unfortunately, to the best of our knowledge, most of the previous methods did not focus on counteracting these challenges explicitly, no matter the analytical methods, or the model-based methods.

In our previous work [117], to help solve these problems, we proposed a novel correlated multi-objective multi-fidelity Gaussian process model (CGP) based on Bayesian optimization. The Bayesian optimization can strike a balance between model accuracies and optimization workloads. Non-linear multi-fidelity models were built to measure the non-linear relationship between the reports of the three design stages and HLS directives. Correlated multi-objective Gaussian process models are proposed as the acquisition functions, to tackle both of the correlated relationships among various design objectives.

Despite that CGP [117] achieved enormous success in modeling the complicated multi-objective and multi-fidelity problem, the shallow structures of Gaussian process models limit the ability to extract information from the input configurations, and bring great challenges to the characterization ability of the kernel functions in the Gaussian process models [132]. Recently, it has been proven that neural networks could automatically discover meaningful representations for the input features by learning multiple layers of highly adaptive basis functions [90, 101, 131–133]. Combining the deep neural networks with GP models can be regarded as the enhancement of the kernel functions, termed as deep kernel functions. The flexibility and automatic calibration provided by the deep kernel functions provide a better performance, with no need for tuning the searching framework for different applications. This technique has achieved more and more attention and applications in the recent few years [101, 131, 133].

We extend our CGP method, by introducing the deep kernel functions to learn better feature representations for the configurations and augment the ability of the Gaussian process models. Our contributions are as follows:

- Bayesian optimization is combined with the multi-fidelity and multi-objective optimization as the optimization framework, to trade-off the model accuracies and optimization workloads.

- Non-linear multi-fidelity models are built to measure the non-linear relationship between the reports of the three design stages and HLS directives. Correlated multi-objective Gaussian process models are proposed, to tackle both of the correlated relationships among various design objectives and the implicit and complicated mapping relationships between directives and objective values, to find Pareto configurations accurately.

- Our method CGP is further enhanced to be a deep version (DCGP) by using deep neural networks as the kernel functions, to learn better feature representations flexibly. Therefore the performance of our method can be further improved significantly.

- Three design objectives, power, delay, and resource consumption are considered in this thesis, thus making the task practical while challenging. We conduct experiments on some public FPGA design benchmarks (including general matrix multiplication and sparse matrix-vector multiplication) and DNN-based object detection model iSmart2. The experimental results show the outstanding performance of our algorithms on the DNN applications and the related applications. The found Pareto designs cover the optimal designs for various design objectives, *i.e.*, power, delay, and resource consumption, and strike a good balance between these objectives.

3.2 Preliminaries

In selecting an optimal HLS directive design, our target is finding a configuration of directives in the space of all directive configurations (named *design space* or *configuration space* in our context) $\mathcal{X}$ which have the optimal performance values in the *objective space* (or *value space*) $\mathcal{Y}$. The design space $\mathcal{X}$ is constructed by enumerating possible HLS directives to be used in the high-level language descriptions. In $\mathcal{X}$, each configuration can be represented as a feature vector $\boldsymbol{x}$. The details on directive encoding are in Section 3.3.2. The objective space $\mathcal{Y}$ is composed of the performance values of the designs given HLS directive configurations. $\mathcal{Y}$ is not known unless we run all of the configurations with the FPGA design tool. Our target is to achieve the configurations with the best performance with no need of knowing the whole objective space $\mathcal{Y}$. In the rest of this chapter, the three FPGA design stages are shorted as *hls*, *syn*, and *impl*.

3.2.1 Bayesian Optimization

Bayesian optimization (BO) is an efficient and widely-used framework [56, 124] to solve global optimization problems, *e.g.*, optimizing analog circuits [85]. For an optimization problem with a black-box objective function f, *e.g.*, power consumption, whose concrete form is unknown, the target of Bayesian optimization is to find a configuration point $\boldsymbol{x} \in \mathcal{X}$ which has the optimal objective value in the objective space $\mathcal{Y}$ by conducting a limited number of trials or evaluations. A two-dimensional example of the mapping from the design space to the objective space is shown in Fig. 3.2.

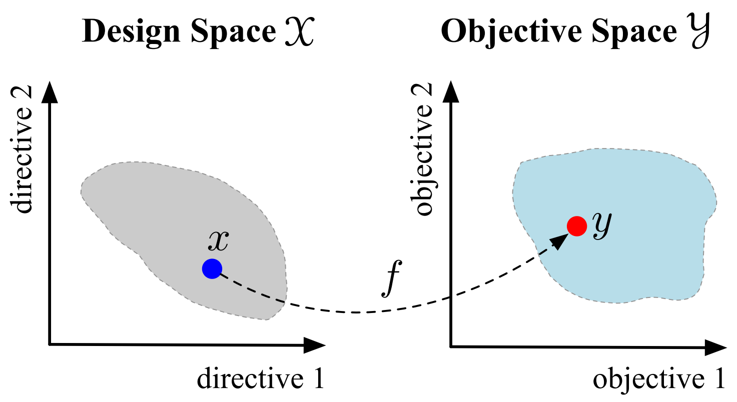


Figure 3.2 An example of mapping from design space $\mathcal{X}$ to objective space $\mathcal{Y}$, with two directives and two design objectives (*e.g.*, power and delay). We need to learn a black-box function f to bridge $\mathcal{X}$ and $\mathcal{Y}$, so as to simulate the FPGA design tool.

In Bayesian optimization, firstly, a set of initial configurations is randomly sampled from the design space $\mathcal{X}$ and passed into the FPGA design tools to get the performance values. These initial data are used to build a *surrogate model* to mimic the objective function. Secondly, the BO algorithm iteratively selects a new configuration from the design space for evaluation under the guidance of an *acquisition function* and then updates the *surrogate model* accordingly. Finally, the optimal HLS directive configuration is the best one explored by the BO algorithm in the optimization process. Three essential elements in Bayesian optimization are as follows:

1) Surrogate model: to optimize the black-box objective function $f : \mathcal{X} \to \mathcal{Y}$, BO learns a probabilistic surrogate model to predict the function value and quantifies the uncertainty of the predictions. A commonly used surrogate model is the Gaussian process (GP) model.

2) Acquisition function $\alpha(\cdot)$ is used as a score function to evaluate the utility of a candidate point $\boldsymbol{x} \in \mathcal{X}$ with respect to finding the optimums of the optimization problem. The acquisition function should balance the exploitation of already-sampled configurations and the exploration of un-sampled configurations in the design space. It is built on the already-sampled configurations and utilizes this prior knowledge (*i.e.*, exploitation) to evaluate the un-sampled configurations (*i.e.*, exploration). There are some popular acquisition functions, such as expected improvement (EI), upper confident bound (UCB), lower confidence bound (LCB), and entropy search (ES) [85].

3) Optimization procedure iteratively samples a configuration from $\mathcal{X}$ based on the acquisition function $\alpha(\cdot)$ in each optimization step and updates the surrogate model accordingly. This process continues until convergency (*i.e.*, no performance improvement in several steps).

3.2.2 Multi-objective Optimization

In the optimization problem of HLS directives, there are multiple objectives to be minimized, *e.g.*, power, delay, and consumption of various types of resources. No matter whether the designers clearly emphasize the single-objective or multi-objective in their problems or not, these multiple objectives should always be considered to guarantee the system's performance. Without loss of generality, our goal is to minimize a group of objectives $f^1(\boldsymbol{x}), f^2(\boldsymbol{x}), \cdots, f^M(\boldsymbol{x}), \forall \boldsymbol{x} \in \mathcal{X}$. Denote the objective values of the $\boldsymbol{x}$ as $\boldsymbol{f}(\boldsymbol{x}) = [f^1(\boldsymbol{x}), f^2(\boldsymbol{x}), \ldots, f^M(\boldsymbol{x})]^\top$. These M objectives would possibly conflict with each other. Finding one solution that minimizes all of these objectives simultaneously is difficult. Practically, to strike a balance between these objectives, we want to identify the Pareto-optimal set.

Definition 1 (Pareto optimality)**.** *In an M-dimension minimization problem, an objective vector $\boldsymbol{f}(\boldsymbol{x})$ is said to dominate $\boldsymbol{f}(\boldsymbol{x'})$ if*

$$\begin{aligned} &\forall i \in [1, M], f^i(\boldsymbol{x}) \leq f^i(\boldsymbol{x'}) \text{ and} \\ &\exists j \in [1, M], f^j(\boldsymbol{x}) < f^j(\boldsymbol{x'}). \end{aligned} \tag{3.1}$$

A point $\boldsymbol{x}$ is Pareto-optimal if there is no other $\boldsymbol{x'}$ in design space satisfying that $\boldsymbol{f}(\boldsymbol{x'})$ dominates $\boldsymbol{f}(\boldsymbol{x})$. In the whole design space, the set of points that are not dominated by others is called the Pareto-optimal set, denoted as $\mathcal{Y}^* \in \mathcal{Y}$. For the Pareto-optimal designs, there does not exist an alternative choice that can improve every objective without sacrificing others. In multi-objective optimization problems, the realistic and accurate goal is to identify the Pareto-optimal set containing all the Pareto-optimal directive configurations. In the previous works on HLS optimization, the M objective functions are solved independently [83], while we proposed to combine them together by a correlation method.

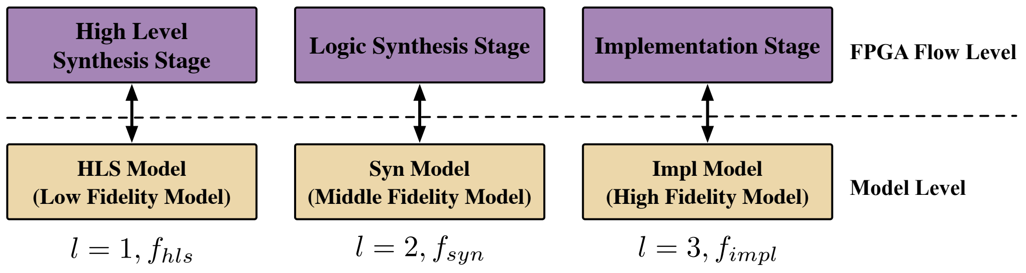
3.2.3 Multi-fidelity Optimization

Definition 2 (Fidelity)**.** *Fidelity refers to the degree to which a model reproduces the state of a real-world project or application. It is therefore a measure of the realism of the model. Straightforwardly, lower fidelity means that the model has lower data accuracy and the higher fidelity is more accurate.*

Definition 3 (Multi-fidelity Model)**.** *For a multi-fidelity problem, each fidelity $l \in \{1, 2, \cdots, L\}$ corresponds to a objective function $f_l(\boldsymbol{x})$. The multi-fidelity model can be defined as:*

$$f_{l+1}(\boldsymbol{x}) = z(f_l(\boldsymbol{x}), \boldsymbol{x}), \tag{3.2}$$

where $z(\cdot)$ is an aggregation function. In our context, $l = 1$ means hls, $l = 2$ means syn, and $l = 3$ means impl.



High Level Synthesis Stage
Logic Synthesis Stage
Implementation Stage
FPGA Flow Level
HLS Model (Low Fidelity Model)
Syn Model (Middle Fidelity Model)
Impl Model (High Fidelity Model)
Model Level
$l = 1, f_{hls}$
$l = 2, f_{syn}$
$l = 3, f_{impl}$

Figure 3.3 The correspondence relationships between the design stages in the FPGA design flow and the multiple models and fidelities in the model level.

We need to define three objective functions for the three stages, *i.e.*, $\{f_{hls}, f_{syn}, f_{impl}\}$. As shown in Fig. 3.1(a), the later stages return more accurate reports, at the cost of consuming longer running times and more computation resources. Therefore, the objective functions for the later stages with more accurate reports would have higher fidelities. The correspondence relationships are shown in Fig. 3.3. There are three models, the HLS model, the Syn model, and the Impl model corresponding to the HLS stage, the Syn stage, and the Impl stage, respectively. That is why it is called multi-fidelity optimization.

In the Bayesian optimization, we define three surrogate models to mimic the objective functions for the three stages, and three acquisition functions to evaluate the utilities of new configurations on these three surrogate models respectively. For convenience, we treat the design *stage* of FPGA design flow and model *fidelity* of algorithms as equivalent and do not distinguish between them, *e.g.*, the lower fidelity implicitly means the lower design stage and vice versa.

3.2.4 Gaussian Process Regression

Gaussian process (GP) regression [96] is a flexible method to model the objective function, which is specified by a mean function $\mu(\boldsymbol{x})$ and a covariance function $k(\boldsymbol{x}, \boldsymbol{x}')$ of the objective $f(\boldsymbol{x})$ as follows:

$$\begin{aligned} \mu(\boldsymbol{x}) &= \mathbb{E}[f(\boldsymbol{x})], \\ k(\boldsymbol{x}, \boldsymbol{x}') &= \mathbb{E}[(f(\boldsymbol{x}) - \mu(\boldsymbol{x}))(f(\boldsymbol{x}') - \mu(\boldsymbol{x}'))]. \end{aligned} \tag{3.3}$$

The mean function $\mu(\boldsymbol{x})$ provides the prior estimations of the objective value for input $\boldsymbol{x}$, and typically a constant mean function $\mu(\boldsymbol{x}) = \mu_0$ is widely used. As to the covariance function

$k(\boldsymbol{x}, \boldsymbol{x}')$, the common form is the squared exponential function of $\boldsymbol{x}$:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \lambda^2 \exp(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}')^\top \boldsymbol{\Lambda}(\boldsymbol{x} - \boldsymbol{x}')), \tag{3.4}$$

where $\boldsymbol{\Lambda} = \mathrm{diag}(\lambda_1^{-2}, \lambda_2^{-2}, \cdots, \lambda_D^{-2})$ is the diagonal length scale matrix, and λ^2 is used to scale the variance of the model.

Define a known single-objective training set $\{\mathcal{X}, \mathcal{Y}\}$, where $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n\}$ is a set of directive configurations and $\mathcal{Y} = \{y_1, y_2, \cdots, y_n\}$ is the corresponding single-objective value set. Assume that the objective function $f(\boldsymbol{x})$ is influenced by the independent and identical zero-mean Gaussian noise $\epsilon_e \sim \mathcal{N}(0, \sigma_e^2)$. Therefore, we have the relationship between directive configuration and its corresponding performance $y_i = f(\boldsymbol{x}_i) + \epsilon_e$, with $i = 1, \cdots, n$.

For a newly sampled configuration $\boldsymbol{x}^*$ and its corresponding objective function f^*, the joint distribution between f^* and the data set $\mathcal{Y}$ which is already sampled in previous steps is defined as follows:

$$p(\mathcal{Y}, f^*) = \mathcal{N}(\begin{bmatrix} \boldsymbol{\mu}_0 \\ \mu_0 \end{bmatrix}, \begin{bmatrix} \boldsymbol{K}(\mathcal{X}) + \sigma_e^2 \boldsymbol{I} & \boldsymbol{k}(\mathcal{X}, \boldsymbol{x}^*) \\ \boldsymbol{k}^\top(\mathcal{X}, \boldsymbol{x}^*) & k(\boldsymbol{x}^*, \boldsymbol{x}^*) \end{bmatrix}), \tag{3.5}$$

where $\boldsymbol{k}(\mathcal{X}, \boldsymbol{x}^*)$ is a vector of covariance values between $\boldsymbol{x}^*$ and all of the configurations in $\mathcal{X}$, and $\boldsymbol{K}(\mathcal{X})$ is the intra-covariance matrix among configurations in $\mathcal{X}$, *i.e.*, $\boldsymbol{K}(\mathcal{X})_{i,j} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ with $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathcal{X}$. According to the Bayes' theorem, the posterior distribution is obtained by:

$$p(f^*|\mathcal{Y}) = \frac{p(\mathcal{Y}, f^*)}{\int p(\mathcal{Y}, f^*)\mathrm{d}f^*} = \mathcal{N}(\mu(\boldsymbol{x}^*), \Sigma(\boldsymbol{x}^*)), \tag{3.6}$$

with

$$\begin{aligned} \mu(\boldsymbol{x}^*) &= \mu_0 + \boldsymbol{k}^\top(\mathcal{X}, \boldsymbol{x}^*)[\boldsymbol{K}(\mathcal{X}) + \sigma_e^2 \boldsymbol{I}]^{-1}(\mathcal{Y} - \boldsymbol{\mu}_0), \\ \Sigma(\boldsymbol{x}^*) &= k(\boldsymbol{x}^*, \boldsymbol{x}^*) - \boldsymbol{k}^\top(\mathcal{X}, \boldsymbol{x}^*)[\boldsymbol{K}(\mathcal{X}) + \sigma_e^2 \boldsymbol{I}]^{-1}\boldsymbol{k}(\mathcal{X}, \boldsymbol{x}^*). \end{aligned} \tag{3.7}$$

Before the posterior is calculated, the hyper-parameters $\boldsymbol{\Lambda}$, λ and σ_e need to be determined by maximum likelihood estimation as follows:

$$\max_{\boldsymbol{\Lambda}, \lambda, \sigma_e} \quad (\mathcal{Y} - \boldsymbol{\mu}_0)^\top(\boldsymbol{K}(\mathcal{X}) + \sigma_e^2 \boldsymbol{I})^{-1}(\mathcal{Y} - \boldsymbol{\mu}_0) + \log|\boldsymbol{K}(\mathcal{X}) + \sigma_n^e \boldsymbol{I}|, \tag{3.8}$$

which can be handled by gradient-based methodologies.

3.2.5 Overview of Our Method

A correlated multi-objective multi-fidelity deep kernel learning GP-based Bayesian Optimization algorithm is proposed to explore the Pareto solutions of HLS directives. The main techniques include the construction of design space, surrogate models and acquisition functions, and optimal configuration selection. In constructing design space, HLS directives are transformed into numerical vectors to perform GP-based Bayesian optimization. Besides, a pruning method is proposed to shrink design space so that the downstream GP-based Bayesian Optimization can explore design space more efficiently. In constructing surrogate models, we combine deep kernel learning with GP to learn better feature representations flexibly. In constructing acquisition functions, we use the expected improvement of Pareto hypervolume as a metric to select the most representative configuration to run the FPGA design tool. Then performance values obtained from the FPGA design tool and the corresponding configuration are used to extend the dataset and update the models. All techniques mentioned above are used to achieve more efficient design space exploration. The brief optimization flow is shown in Fig. 3.4. In Section 3.5, the overall detailed optimization flow and algorithm framework are provided to solve the optimization problems of HLS directives.

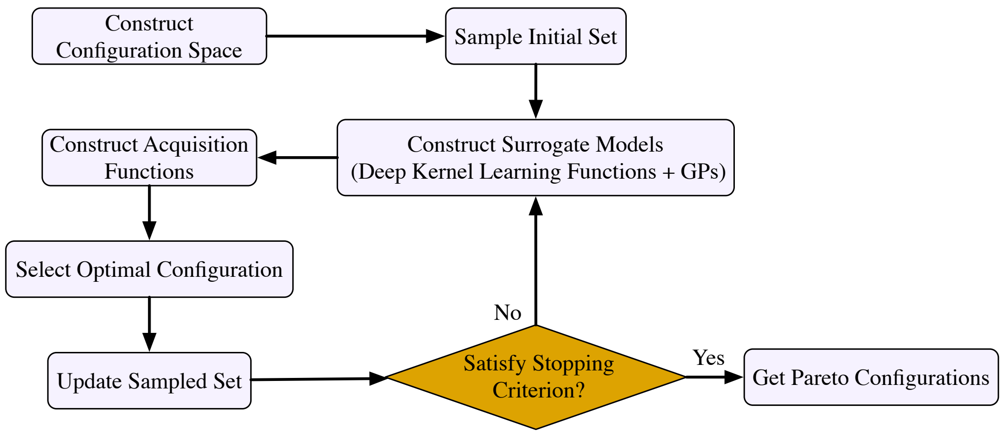

Construct Configuration Space
Sample Initial Set
Construct Acquisition Functions
Construct Surrogate Models (Deep Kernel Learning Functions + GPs)
Select Optimal Configuration
No
Update Sampled Set
Satisfy Stopping Criterion?
Yes
Get Pareto Configurations

Figure 3.4 The brief flow of our method for HLS directives optimization.

3.3 HLS Directive Design Space

To construct the design space $\mathcal{X}$, we enumerate the designs, conduct the design space pruning to remove infeasible configurations and encode the directive configurations as feature vectors.

3.3.1 Design Space and Space Pruning

Some HLS directives are widely used, including pipelining, loop unrolling, and array partitioning, *etc.* These directives are especially popular in DNN-based applications which are composed of many dense matrix operations. Some typical dense matrix operations include fully connected operations, convolutional operations, and general matrix multiplication (`GEMM`). In general applications, the codes are composed of several for-loops, some arrays, and related computations. The design space can be generated by direct permutations and combinations of directives.

However, some directives are conflicting and some are obviously non-optimal, especially for loop unrolling and array partitioning. Infeasible configurations may increase optimization workloads. For example, for an array used in a for-loop, if the array partitioning factor is less than the loop unrolling factor, this loop may not be unrolled successfully because the visits to the array are limited by the partitioning factor [4]. If the array partitioning factor is greater than the loop unrolling factor, more memory resources are consumed without increasing the system parallelism. Under this circumstance, compatible directives and factors are the best. A design space pruning method is proposed to help solve this problem, as shown in Algorithm 1. The inputs are high-level language descriptions and a file that indicates which directives are to be analyzed.

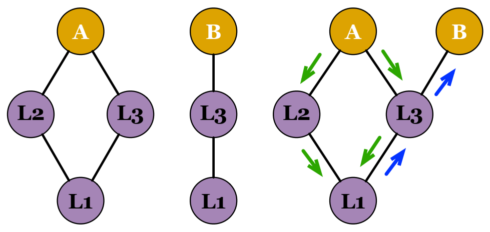
Fig. 3.5 shows an example. There are two arrays A and B,

Algorithm 1 The Pseudo-code of Pruning Method

```
1: Inputs: High-level programming language source code, and a directive file;
2: Outputs: Pruned design space $\mathcal{X}$, initially $\mathcal{X} \leftarrow \emptyset$;
3: Construct a graph for each array, with itself as the root node and related loops as children nodes;
4: Merge graphs with common nodes, denote the set of graphs as $\mathcal{T}$;
5: for all graph $t_i \in \mathcal{T}$ do
6:     for root (array) node $a_j$ in $t_i$ do
7:         for partitioning factor $f_k$ of $a_j$ do
8:             Assign $f_k$ to $a_j$;
9:             Assign a unrolling factor to each loop node in $t_i$;
10:            Backtrack from leaf nodes, assign partitioning factors to array nodes in $t_i$, except $a_j$;
11:        end for
12:        Record feasible configurations of $a_j$ as set $C_j$;
13:        $\mathcal{X} \leftarrow \mathcal{X} \cup C_j$;
14:    end for
15: end for
16: Traverse $\mathcal{X}$ and remove repeated configurations;
17: return Pruned design space $\mathcal{X}$.
```

```
for L1 in range(0,N1):
 for L2 in range(0,N2):
  op(A[ L1 * 10 + L2 ])
 for L3 in range(0,N3):
  op(A[ L1 * 10 + L3 ])
  op(B[ L1 * 10 + L3 ])
```

(a)



(b)

Figure 3.5 An example of design space pruning method. (a) Code with three loops and two arrays. L1 is the outer loop. L2 and L3 are the inner parallel loops. (b) Graphs of array A and B, and the merged graph.

and three loops L1, L2, and L3 in Fig. 3.5(a). Two graphs are built for A and B, with arrays as root nodes and loops as non-root nodes, as shown in Fig. 3.5(b). The outer loop L1 is the leaf node and nested loops L2 and L3 are non-leaf nodes. These two graphs are merged since they share some common nodes L3 and L1. To help understand the common nodes of the two arrays, we build two graphs separately and then merge them. Merging trees can also be finished while constructing the graphs for arrays and there is no effect on the results. In each graph, the factor of each child node is determined by its parents. Two types of array partitioning are considered here, `CYCLIC` and `BLOCK`, as shown in Fig. 2.2.

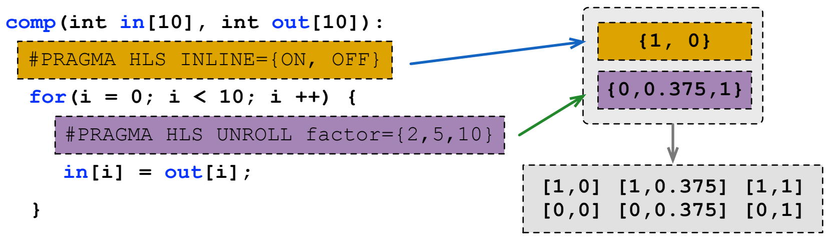
If we partition A with type `CYCLIC`, then we will assign unrolling factors for L2 and L3. But we will not unroll L1. In other words, the unrolling factor is 1, because L1 is incompatible with `CYCLIC` partitioning of A and unrolling of L2 and L3. After that, we will backtrack from L1 to assign `CYCLIC` partitioning factors to B because A and B are in the same loop L3 and their partitioning types should be the same. In the graph, they are connected by the common node L3.

If we partition array A with type `BLOCK`, we will set the unrolling factors of L2 and L3 as 1. The reason is that the unrolling of these two loops is incompatible with `BLOCK` of A. But we can unroll loop L1 successfully because they are compatible. After that, we will backtrack from L1 to B, to partition array B with type `BLOCK`.

In this process, all factors will be checked whether they are compatible, and more domain knowledge can be used here if wanted. All of the compatible configurations are added into a configuration set C_{A} belonging to A. After identifying C_{A} and adding C_{A} into $\mathcal{X}$, we can conduct the same configuration assignment process starting at array B in the merged graph. Note that finally, we will traverse $\mathcal{X}$ again to remove repeated configurations. The invalid and incompatible directive configurations are pruned with the graph-based method, which also eases the optimization task. For the applications with extremely large design spaces, sampling techniques can be adopted, *e.g.*, Monte Carlo-based sampling with reparameterization trick [134, 135]. Therefore, our proposed method can be used to explore large design spaces efficiently.

3.3.2 Encoding of Directive Configurations

The GP, as a typical machine learning model, can only work on numerical values. Therefore, it is necessary to transform the non-numerical design parameters, such as unroll and inline, into numerical arrays. The TRUE/FALSE factors are represented as 0 or 1 directly. The directives which have several factors are represented as normalized features, *e.g.*, three factors $\{2, 5, 10\}$ are encoded as $\{0, 0.375, 1\}$. The normalization can adjust all features to the same scale and allows for a more uniform influence for all weights and faster convergence on learning [16]. If the partitioning factor 2 has good performances, we will conjecture that 5 is better than 10 because 5 is closer to 2. Fig. 3.6 shows an example. In this example, there are two HLS directives and six configurations in the design space. More directives with factors are included in the experiments, *e.g.*, pipelining and array partitioning. The final feature vector for a code segment is the concatenations and combinations of features of all the directives in this segment.



```
comp(int in[10], int out[10]):
 #PRAGMA HLS INLINE={ON, OFF}
  for(i = 0; i < 10; i ++) {
     #PRAGMA HLS UNROLL factor={2,5,10}
     in[i] = out[i];
  }
```

```
{1, 0}
{0,0.375,1}

[1,0] [1,0.375] [1,1]
[0,0] [0,0.375] [0,1]
```

Figure 3.6 An example of directive encoding.

3.4 Correlated Multi-objective Multi-fidelity Models and Deep Kernel Functions

In this section, non-linear multi-fidelity models and correlated multi-objective models are described in Section 3.4.1 and Section 3.4.2 and are enhanced by deep kernel functions in Section 3.4.4.

3.4.1 Non-linear Multi-Fidelity Model

Traditionally, in HLS directive designs, the relationship among the multiple stages (fidelities) is assumed to be linear. For example, in [83], higher fidelity estimates scale the lower fidelity output by a factor and add an independent GP to model the remaining differences. However, it is not suitable and weak for the applications where these three FPGA design stages exhibit strong complicated correlations. Therefore, non-linear models are proposed to further exploit the corresponding non-linear relationships between the low- and high-fidelity objective functions. The non-linear model can be formulated as Equation (3.9).

$$f_{i+1}(\boldsymbol{x}) = z(f_i(\boldsymbol{x}), \boldsymbol{x}) + f_e(\boldsymbol{x}), \forall i \in \{1, \dots, L-1\}, \tag{3.9}$$

where $z(\cdot)$ is the non-linear function and is modeled by a GP model, and $f_e(\boldsymbol{x})$ is the error term which is also defined as a GP model. The outputs $f_i(\boldsymbol{x})$ of the early stage (low Fidelity) model are concatenated with the directive encoding features $\boldsymbol{x}$ as the input features to the later stage (high fidelity) GP model.

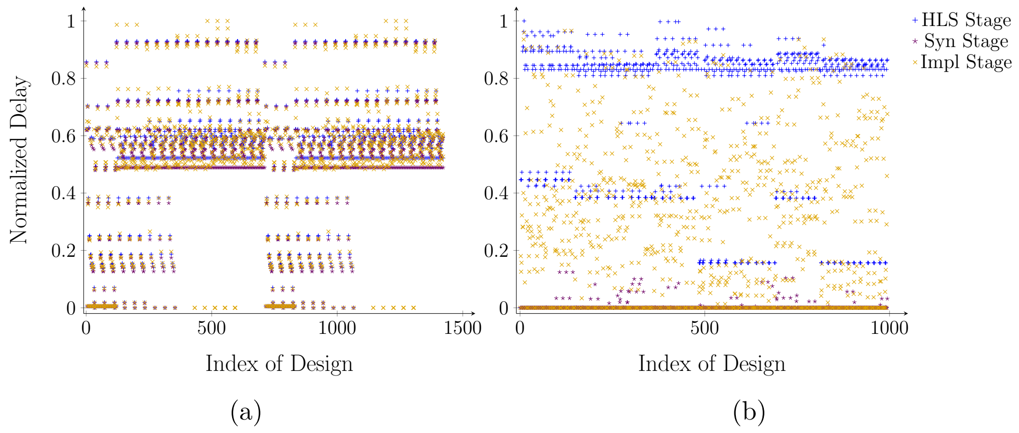


Figure 3.7 Normalized delay values of the three fidelities. The X-axis is the index of the design. We assign indices for these designs according to their directive values in increasing order. (a) `GEMM` (general matrix multiplication). (b) `SPMV_ELLPACK` (sparse matrix-vector multiplication using the ELLPACK format).

Delay values of two benchmarks are shown in Fig. 3.7 as examples to illustrate the complex non-linear relationships among the three fidelities. In the general matrix multiplication (`GEMM`), delay values of the configurations in the three fidelities are highly overlapping. For the sparse matrix-vector multiplication using the `ELLPACK` format (`SPMV_ELLPACK`), delay values in the three fidelities show high divergences. The high divergences of various applications make it hard to regress the relationships accurately by using traditional linear models. Obviously, using non-linear models is a wise and general choice to handle various applications.

3.4.2 Correlated Multi-Objective Model

To learn and measure the Pareto set for the multi-objective optimization problem, we introduce the expected improvement of Pareto hypervolume (EIPV) [105] and define it as the acquisition function. Firstly, we will clarify the concept of the expected improvement of Pareto hypervolume. Secondly, we will define the probability model and compute the value of the expected improvement.

Assume that in current optimization step $t+1$, we already have a Pareto-optimal set $\mathcal{D} = \{\mathcal{X}^*, \mathcal{Y}^*\}$, with $\mathcal{X}^* = \{\boldsymbol{x}_s\}_{s=1}^t$, and $\mathcal{Y}^* = \{\boldsymbol{y}_s\}_{s=1}^t$. Note that $\mathcal{D}$ is the Pareto-optimal set of the designs explored in the previous t steps. A virtual configuration point $\boldsymbol{v}_{ref} \in \mathbb{R}^M$ is defined as the reference point, which is dominated by $\mathcal{Y}^*$, *i.e.*, $\boldsymbol{y}_s \succeq \boldsymbol{v}_{ref}$ [1] for $\forall \boldsymbol{y}_s \in \mathcal{Y}^*$. $\boldsymbol{v}_{ref}$ does not have physical meanings and is only for the ease of computations. In the experiments, we can directly assign extremely large values which usually do not occur in practical scenarios to $\boldsymbol{v}_{ref}$, *e.g.*, 100W for power. The Pareto hypervolume with respect to $\boldsymbol{v}_{ref}$ in the objective space is defined as Equation (3.10).

$$\mathrm{PV}_{\boldsymbol{v}_{ref}}(\mathcal{Y}^*) = \int_{\mathbb{R}^M} \mathbb{I}[\boldsymbol{y} \succeq \boldsymbol{v}_{ref}] \left[1 - \prod_{\boldsymbol{u} \in \mathcal{Y}^*} \mathbb{I}[\boldsymbol{u} \not\succeq \boldsymbol{y}]\right] \mathrm{d}\boldsymbol{y}, \quad (3.10)$$

where $\mathbb{I}(\cdot)$ is the indicator function, which outputs 1 if its argument is true and 0 otherwise. This equation measures the

[1] "$\succeq$" denotes "dominate". In this minimization problem, its numerical meaning is "$\leq$" as shown in Equation (3.1).

volume of the objective space composed of configurations which dominate $\boldsymbol{v}_{ref}$ but are dominated by at least one configuration in $\mathcal{Y}^*$. The greater the volume is, the better the Pareto set is.

Greedily, in each operation step, we want to sample a configuration which can lead to the highest expected improvement of the Pareto hypervolume. Here the "expected" comes from the uncertainty information of the predicted performance values of GP models. We will estimate the expected improvements for the un-sampled configurations and select the configuration which leads to the largest expected improvement. The expected improvement is defined as Equation (3.11).

$$\mathrm{EIPV}(\boldsymbol{x}_{t+1}|\mathcal{D}) = \mathbb{E}_{p(\boldsymbol{y}(\boldsymbol{x}_{t+1})|\mathcal{D})}\left[\mathrm{PV}_{\boldsymbol{v}_{ref}}\left(\mathcal{Y}^* \cup \boldsymbol{y}(\boldsymbol{x}_{t+1})\right) - \mathrm{PV}_{\boldsymbol{v}_{ref}}\left(\mathcal{Y}^*\right)\right]. \tag{3.11}$$

An example of the Pareto hypervolume is shown in Fig. 3.8. The objective space is divided into cells according to the locations of the currently found Pareto set. Orange points are Pareto points and blue points are dominated. Blank cells are dominated while light yellow cells are not. Volume of the blank cells is the current Pareto hypervolume. We can decompose the whole objective space into grid cells to simplify the integration of Equation (3.10), as shown in Fig. 3.8(a). The decomposition is according to the locations of the found Pareto-optimal configurations in the objective space. The corresponding objective values at these two axes are b_i^1 and b_i^2. We denote the non-dominated cells as $\mathcal{C}_{\mathrm{nd}}$. Then Equation (3.11) is simplified as

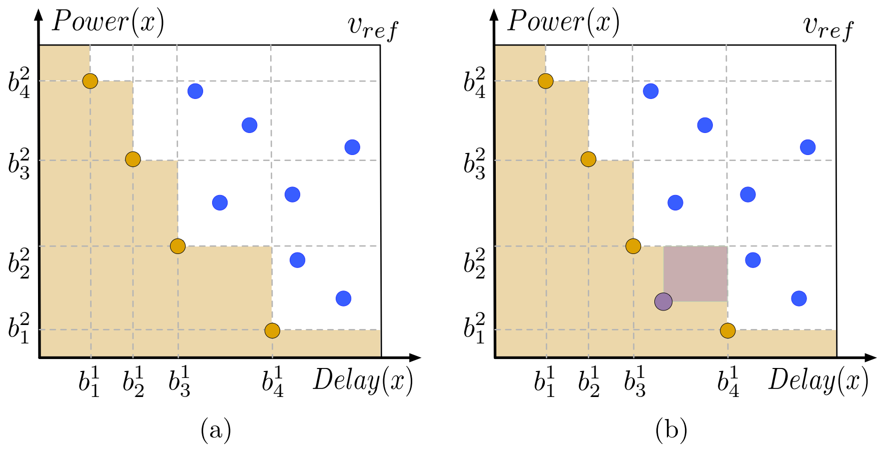


Figure 3.8 An example of minimizing power and delay.

Equation (3.12), where $\Delta_C(\boldsymbol{x})$ is the volume of cell $C \in \mathcal{C}_{\mathrm{nd}}$.

$$\mathrm{EIPV}(\boldsymbol{x}_{t+1}|\mathcal{D}) = \sum_{C \in \mathcal{C}_{\mathrm{nd}}} \Delta_C(\boldsymbol{x}) = \sum_{C \in \mathcal{C}_{\mathrm{nd}}} \int_C \mathrm{PV}_{\boldsymbol{v}_c}(\boldsymbol{y}) p(\boldsymbol{y}|\mathcal{D}) \mathrm{d}\boldsymbol{y}. \tag{3.12}$$

In Fig. 3.8(b), purple point minimizes the expected improvement and is predicted to be the Pareto-optimal configuration. The light purple cell is the corresponding expected improvement of Pareto hypervolume.

Now we have clarified the concept of the expected improvement of Pareto hypervolume. The next step is to define the probability model $p(\boldsymbol{y}|\mathcal{D})$ and to further deduce the concrete form of the expected improvements. In previous works [82, 83], the multiple design objectives are predicted via several independent Gaussian process models, though in real applications, they are usually correlated. For example, to reduce system delay, we may want to increase the system parallelism which means we will consume much more on-chip resources, *e.g.*, LUTs. Therefore, delay and resource consumption are negatively correlated. But power and resource consumption are positively correlated since that instantiating more on-chip resources would increase power consumption simultaneously.

In this thesis, $p(\boldsymbol{y}|\mathcal{D})$ is modeled as a correlated multi-objective GP model [15], as shown in Equation (3.13).

$$p(\boldsymbol{y}|\mathcal{D}) = \mathcal{N}(y^1, \ldots, y^M; \boldsymbol{\mu}, \boldsymbol{\Sigma}), \tag{3.13}$$

where $\boldsymbol{\mu}$ is the mean vector with length M and each element μ_i in it is the mean value of objective f^i. The covariance matrix $\boldsymbol{\Sigma}$ is non-diagonal. Specifically, definition of the covariance value is:

$$\boldsymbol{\Sigma}_{i,j} = \mathrm{Cov}(f^i(\boldsymbol{x}),\ f^j(\boldsymbol{x}')) = \mathrm{K}_{i,j} k_C(\boldsymbol{x}, \boldsymbol{x}'), \tag{3.14}$$

where $\mathrm{K}_{i,j}$ is the similarity between objectives i and j and can be obtained by maximizing likelihood estimation. k_C is a covariance function over $\mathcal{X}$ and is defined as automatic relevance determination (ARD) Matérn 5/2 kernel to avoid over-smoothness [109].

3.4.3 Combined Model

Our method has two novel modeling techniques, one for modeling the multiple correlated objectives and one for modeling the three fidelities (*i.e.*, three FPGA design stages). At each fidelity, all of the objectives construct a correlated multi-objective model. Its expected improvement function of Pareto hypervolume is denoted as $\mathrm{EIPV}_i(\boldsymbol{x}_{t+1}|\mathcal{D})$, with $i \in \{hls, syn, impl\}$. In the real FPGA design flow, obtaining results in different stages costs different running times. To characterize the different costs, an additional penalty term ρ_i is applied to augment $\mathrm{EIPV}_i(\boldsymbol{x}_{t+1})$ as the penalized EIPV, termed as $\mathrm{PEIPV}_i(\boldsymbol{x}_{t+1}|\mathcal{D})$:

$$\begin{aligned}&\mathrm{PEIPV}_i(\boldsymbol{x}_{t+1}|\mathcal{D}) = \rho_i \cdot \mathrm{EIPV}_i(\boldsymbol{x}_{t+1}|\mathcal{D}),\\ &\rho_i = \frac{T_{impl}}{T_i}, i \in \{hls, syn, impl\},\end{aligned} \tag{3.15}$$

where T_i is the time of running the FPGA design tool from scratch to stage i. Finally, PEIPV functions in Equation (3.15) are used as the acquisition functions in the Bayesian optimization framework. For the designs that violate the design rules, no valid reports are returned from the FPGA tool. Their simulation performance is set to be $10\times$ worse than the current worst-case, to punish the illegal designs and teach the models. Fig. 3.9 visualizes the structures of the combined models. The orange lines represent the non-linear relationships. The blue lines represent the inputs. Each fidelity has an acquisition function PEIPV.

3.4.4 Optimization Based on Deep Kernel Functions

The properties of the distributions over functions induced by a GP are controlled by the kernel function, and the covariance matrices implicitly depend on the hyper-parameters in the kernel functions [34, 64, 132, 133, 140]. From this perspective, learning better kernel functions is of vital importance to the performance of the GP models. In the recent fewer years, deep neural networks have been shown to have powerful mechanisms to create

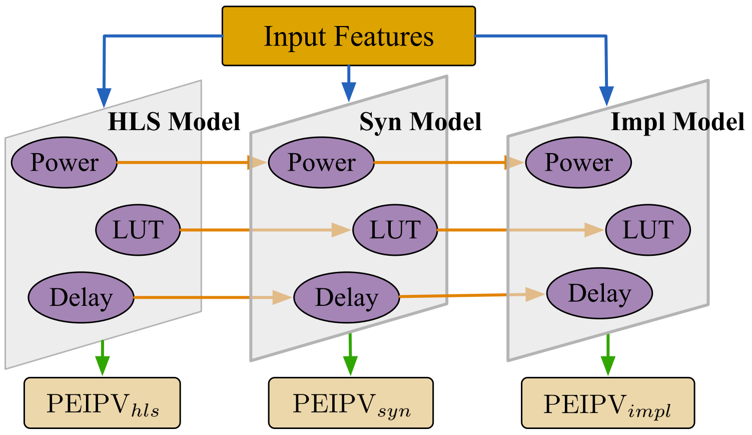


Figure 3.9 The combined models, with three stages (fidelities) and three objectives.

adaptive functions to discover meaningful representations of input data. Therefore, we propose to use the deep neural network as the deep kernel function in the GP models.

As mentioned above, the covariance value in the covariance matrix Σ is defined as Equation (3.14), where k_C is defined as ARD Matérn 5/2 kernel. ARD Matérn 5/2 kernel [97] takes the form:

$$k_C(\boldsymbol{x}, \boldsymbol{x}') = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu d}\right) K_\nu \left(\sqrt{2\nu d}\right), \quad d = (\boldsymbol{x} - \boldsymbol{x}')^\top \theta^{-1} (\boldsymbol{x} - \boldsymbol{x}'), \tag{3.16}$$

where Γ is the gamma function, ν is the smoothness parameter ($\nu = 5/2$), K_ν is a modified Bessel function of the second kind, and θ is the parameters to be learned. Despite the complicated form, it can be regarded as the inner product of the input feature vectors. Since the covariance only depends on the distances between inputs, it is stationary. In different applications, it is hard to give a general and uniform representation for the different feature vectors to characterize the complicated relationships between the design configurations.

To improve the characterization ability of the kernel function, we propose to use deep neural networks to enhance the kernel function, termed as deep kernel function, as shown in Equation (3.17).

$$k'_C(\boldsymbol{x}, \boldsymbol{x}') = k_C(\phi_W(\boldsymbol{x}), \phi_W(\boldsymbol{x}')|\theta, W), \tag{3.17}$$

where $\phi_W(\cdot)$ represents the neural network and W represents the parameters in the network. Implicitly, that is equivalent to learn a novel distance metric to enhance the distance d in Equation (3.16), *i.e.*,

$$d' = (\phi_W(\boldsymbol{x}) - \phi_W(\boldsymbol{x}'))^\top \theta^{-1}(\phi_W(\boldsymbol{x}) - \phi_W(\boldsymbol{x}')). \tag{3.18}$$

The structure of the model with deep kernel function is shown in Fig. 3.10. The original input feature is $\boldsymbol{x}$. And the learned novel feature representation is $\phi_W(\boldsymbol{x})$. The learned feature of the input configuration is $\phi_W(\boldsymbol{x})$. Then $\phi_W(\boldsymbol{x})$ is used as the inputs to the GP models. The structure of our deep kernel function is described in detail in the experiments.

The weight W of the neural network is a part of the parameters in our method. For brevity, denote the multi-objective GP model as GP_Θ with parameters Θ. Parameters Θ and W are optimized jointly, by maximizing the log marginal likelihood $\mathcal{L}$ of the Gaussian process model. According to the chain rule of

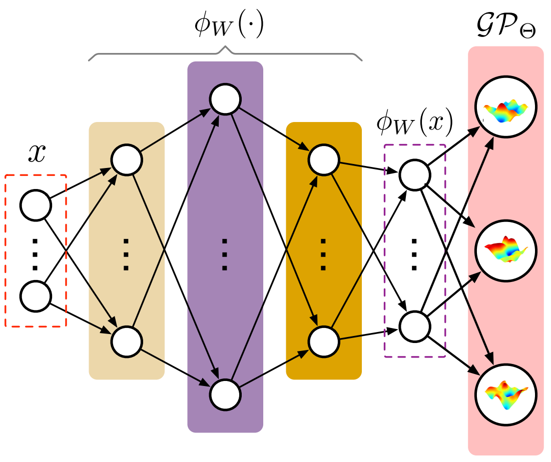


Figure 3.10 Our method combines the GP model GP_Θ and the deep kernel function ϕ_W.

the gradients, Θ and W can be updated according to:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \Theta} &= \frac{\partial \mathcal{L}}{\partial \mathrm{K}_C}\frac{\partial \mathrm{K}_C}{\partial \Theta},\\ \frac{\partial \mathcal{L}}{\partial W} &= \frac{\partial \mathcal{L}}{\partial \mathrm{K}_C}\frac{\partial \mathrm{K}_C}{\partial \phi_W(\boldsymbol{x})}\frac{\partial \phi_W(\boldsymbol{x})}{\partial W},\end{aligned} \tag{3.19}$$

where K_C denotes the kernel functions which contain $\mathrm{K}_{i,j}$ in Equation (3.14) and k'_C in Equation (3.17), $\frac{\partial \mathrm{K}_C}{\partial \Theta}$ represents the derivatives of the kernel with respect to the kernel parameters. $\frac{\partial \mathrm{K}_C}{\partial \phi_W(\boldsymbol{x})}$ represents the derivatives of the deep kernel with respect to the neural network ϕ_W, while Θ is fixed. With this training method, all of the parameters are trained jointly according to a unified supervised objective, as part of the Gaussian process framework, without requiring approximate Bayesian inference. During training, the gradients are computed via back propagation.

3.5 The Overall Optimization Flow

The Bayesian optimization method is adopted as the algorithm skeleton to explore the Pareto-optimal directive configurations, with the GP models as the surrogate models, and PEIPV functions as the acquisition functions.

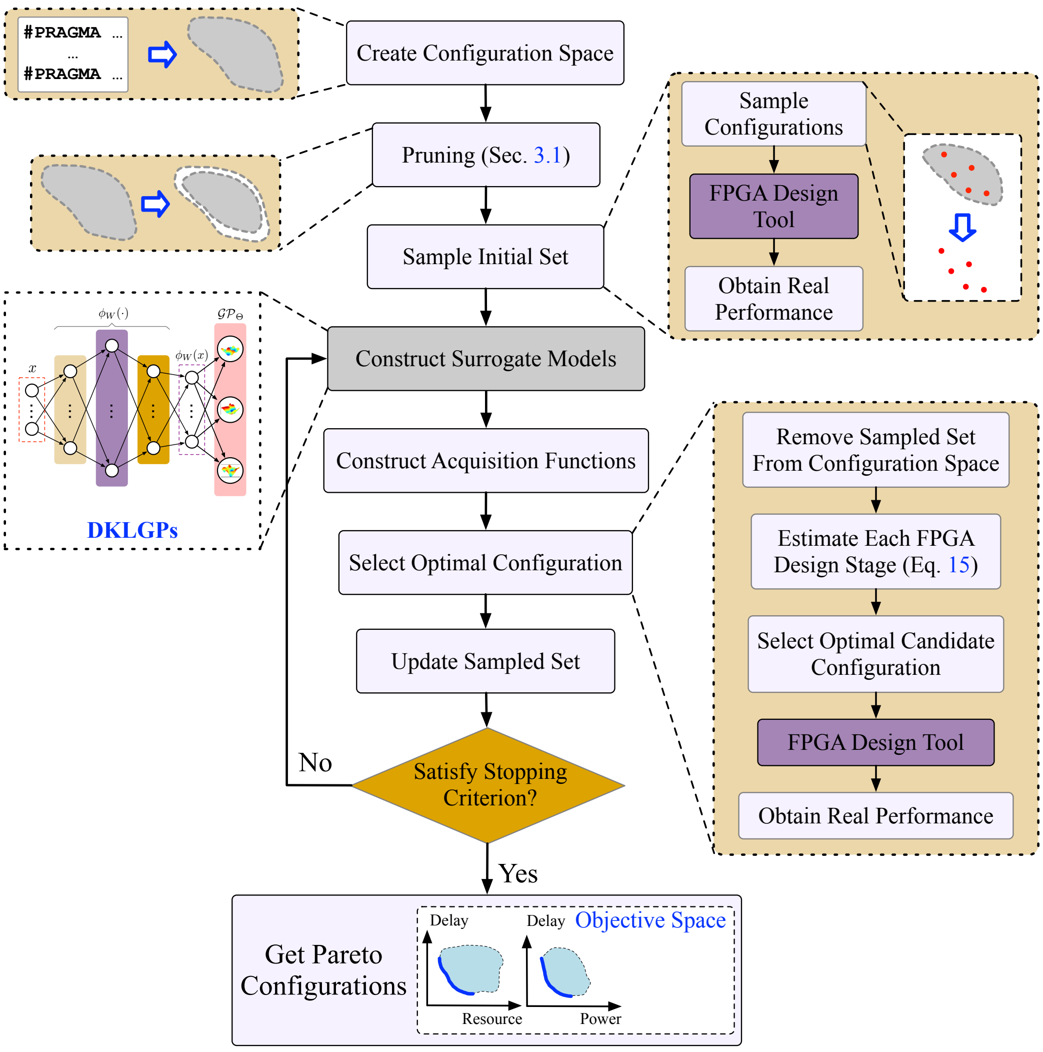

#PRAGMA ...
...
#PRAGMA ...
Create Configuration Space
Pruning (Sec. 3.1)
Sample Initial Set
Sample Configurations
FPGA Design Tool
Obtain Real Performance
Construct Surrogate Models
DKLGPs
Construct Acquisition Functions
Select Optimal Configuration
Remove Sampled Set From Configuration Space
Estimate Each FPGA Design Stage (Eq. 15)
Select Optimal Candidate Configuration
FPGA Design Tool
Obtain Real Performance
Update Sampled Set
Satisfy Stopping Criterion?
No
Yes
Get Pareto Configurations
Objective Space
Delay
Resource
Delay
Power

Figure 3.11 Detailed overall optimization flow.

Using data from later stages contributes to a more accurate surrogate model, at the cost of more simulation workloads to obtain the performance values. If the results at the early FPGA design stage are good enough, there is no need to run the FPGA design tool to the later stages. In each optimization step of the BO algorithm, for the surrogate model of each stage, we need to consider the quality of the selected point and its acquisition value, so as to determine whether it is necessary to optimize models of the later stages. For example, in one BO step, PEIPV_{syn} is the best compared with PEIPV_{hls} and PEIPV_{impl}, at configuration $\boldsymbol{x}_{syn}$. We will then run the FPGA design tool with $\boldsymbol{x}_{syn}$ as the directive configuration input, to get the real performance values at *hls* and *syn* stages. Finally, we will update the surrogate models of *hls* and *syn* stages according to these performance values. If PEIPV_{impl} is the best, we will run the FPGA design tool to the final *impl* stage, and update the models of *hls*, *syn*, and *impl* stages.

The overall optimization flow is detailed in Algorithm 2 and Fig. 3.11. DKLGP denotes the deep kernel learning functions and Gaussian process model. Firstly, we define and prune the design space according to the tree-based method described in Algorithm 1. Denote the generated design space as $\mathcal{X}$. Secondly, we randomly sample some configurations from the design space for initialization. The configurations for the higher fidelities (later FPGA stages) are subsets of the lower fidelities (earlier FPGA stages), *i.e.*, $\mathcal{X}_{impl} \subseteq \mathcal{X}_{syn} \subseteq \mathcal{X}_{hls} \subseteq \mathcal{X}$. These configurations are then fed into the FPGA design tool to get

Algorithm 2 The Bayesian Optimization-based Optimization Flow

```
1: **Inputs**: High-level programming language source code, optimization steps $N_{iter}$, early-stopping step $S_E$;
2: **Outputs**: Pareto configuration set $\mathcal{X}^*$ and objective value set $\mathcal{Y}^*$, initially $\mathcal{X}^* \leftarrow \emptyset$, $\mathcal{Y}^* \leftarrow \emptyset$;
3: Enumerate the design space and run tree-based pruning method, to get pruned design space $\mathcal{X}$;   ▷ Section 3.3.1
4: Randomly sample initial sets $\mathcal{X}_{impl} \subseteq \mathcal{X}_{syn} \subseteq \mathcal{X}_{hls} \subseteq \mathcal{X}$;
5: Run FPGA tool to get the performance values $\mathcal{Y}_i$ for $\mathcal{X}_i$, with $i \in \{hls, syn, impl\}$;
6: Initialize a surrogate model DKLGP$_i$ and an acquisition function PEIPV$_i$ for each stage $i$ according to $\{\mathcal{X}_i, \mathcal{Y}_i\}$, with $i \in \{hls, syn, impl\}$;   ▷ DKLGP$_i$ is our method with GPs and deep kernel learning functions
7: **for** $t \leftarrow 1$ to $N_{iter}$ **do**
8:     **for all** stage $i \in \{hls, syn, impl\}$ **do**
9:         Update DKLGP$_i$ and PEIPV$_i$ according to $\{\mathcal{X}_i, \mathcal{Y}_i\}$;
10:        $\hat{\boldsymbol{x}}_i \leftarrow \arg\max_{\boldsymbol{x} \in \mathcal{X}}$ PEIPV$_i(\boldsymbol{x})$;   ▷ Select the candidate Pareto configuration from $\mathcal{X}$
11:    **end for**
12:    $(\boldsymbol{x}^*, h) \leftarrow \arg\max_{(\hat{\boldsymbol{x}}_i, i)}$ PEIPV$_i(\boldsymbol{x})$, with $i \in \{hls, syn, impl\}$;   ▷ Determine the Pareto configuration
13:    Run FPGA tool with $\boldsymbol{x}^*$ up to stage $h$, to get $\boldsymbol{y}_i$, with $i \in \{hls, ..., h\}$;
14:    $\mathcal{X}_i \leftarrow \mathcal{X}_i \cup \boldsymbol{x}^*$, $\mathcal{Y}_i \leftarrow \mathcal{Y}_i \cup \boldsymbol{y}_i$, with $i \in \{hls, ..., h\}$;
15:    $\mathcal{X} \leftarrow \mathcal{X} \setminus \boldsymbol{x}^*$;   ▷ Remove $\boldsymbol{x}^*$ from the design space
16:    Compute current hypervolume $hv_t$ of $\{\mathcal{X}_{impl}, \mathcal{Y}_{impl}\}$;
17:    **if** $hv_t$ has no improvements in the continuous $S_E$ steps **then**   ▷ Early-stopping condition
18:        break;   ▷ Converge and exit the optimization process
19:    **end if**
20: **end for**
21: Select Pareto configurations $\{\mathcal{X}^*, \mathcal{Y}^*\}$ from $\{\mathcal{X}_{impl}, \mathcal{Y}_{impl}\}$;   ▷ Obtain the final results from the exploration record
22: **return** Pareto configurations $\{\mathcal{X}^*, \mathcal{Y}^*\}$.
```

real performance values $\mathcal{Y}_i$, with $i \in \{hls, syn, impl\}$. For each stage, we initialize a surrogate model DKLGP$_i$ (*i.e.*, GP model with deep kernel learning function) and an acquisition function PEIPV$_i$. In each optimization time step, for each stage i, we will select a configuration $\hat{\boldsymbol{x}}_i \in \mathcal{X}$ which maximizes the expected improvement PEIPV$_i$. $\hat{\boldsymbol{x}}_i$ is regarded as the candidate Pareto configuration of this stage. Then a node-stage pair $(\boldsymbol{x}^*, h)$ which achieves the highest expected improvement is selected from the three $\hat{\boldsymbol{x}}_i$ configurations. Here h denotes the stage index. $\boldsymbol{x}^*$ is our final choice of Pareto point in current optimization step. A toy example on the surrogate models and PEIPV functions is shown in Fig. 3.12. Red nodes are the sampled configurations. Models of the lower stages have wider error ranges (light yellow fillers) since their fidelities are lower. Each stage selects a configuration with the highest expected improvement, *i.e.*, x_1, x_2, and x_3. x_1 has the higher expected improvement compared with x_2 and x_3. Therefore, in this optimization step, the HLS stage is selected and x_1 is sampled. We will pass the code together with configuration $\boldsymbol{x}^*$ into the FPGA design tool, run the tool up to stage h to get the performance values (*i.e.*, $\boldsymbol{y}_i$, with $i = hls, ..., h$). Record the configuration and performance values, *i.e.*, $\mathcal{X}_i \leftarrow \mathcal{X}_i \cup \{\boldsymbol{x}^*\}$, and $\mathcal{Y}_i \leftarrow \mathcal{Y}_i \cup \{\boldsymbol{y}_i\}$, and update all of the corresponding surrogate models and PEIPV functions. Then we will start the next BO searching step. The final Pareto designs $\{\mathcal{X}^*, \mathcal{Y}^*\}$ found by our optimization method are computed from

$\{\mathcal{X}_{impl}, \mathcal{Y}_{impl}\}$ [2].

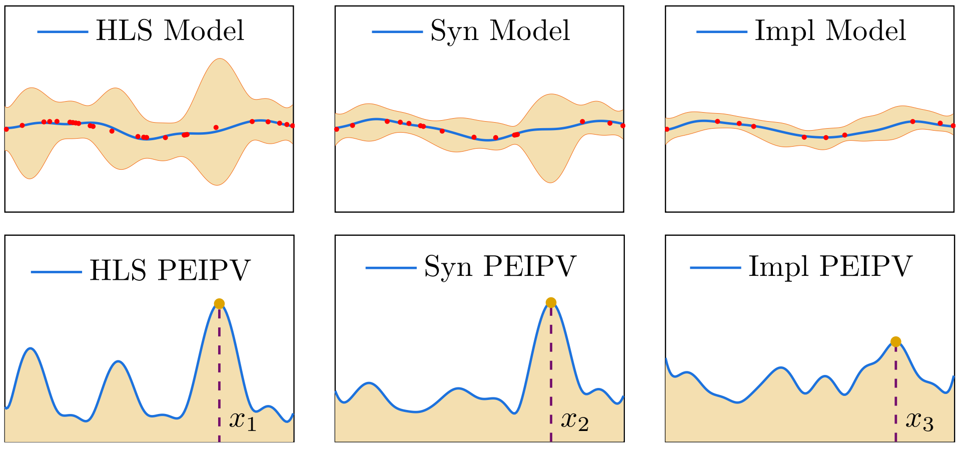


Figure 3.12 A toy example to explain the models of the 3 stages (as shown in Fig. 3.3) and their corresponding acquisition functions.

Note that in our framework, no additional model training dataset is required. Some configurations are sampled from the design space to initialize the models during the model initialization, *i.e.*, train the model from scratch. These initial configurations with their performance values $\{\mathcal{X}_i, \mathcal{Y}_i\}, i \in \{hls, syn, impl\}$ are the initial training set. Then, new configurations are selected from the design space in the iterative optimization process, according to the expected improvements discussed above. These newly sampled configurations are passed to the FPGA design tool to get actual performance values and extend the training set to tune the model further. The deep kernel modules and GP modules are trained jointly according to Equation (3.19). The process is repeatedly performed several times until convergence.

[2]Note that different from the optimization process, given a set with the known objective values, the Pareto set of this set is deterministic and can be computed easily.

Compared with our previous work CGP [117], considering that the shallow structures of Gaussian process models limit the ability to extract information from the input configurations, we combine the deep kernel learning functions with GP models in DCGP, as the enhancement of the kernel functions to learn better feature representations flexibly. Based on this combination, the proposed DCGP is expected to further improve significantly the performance.

Our proposed method can be easily used to explore large design spaces. If the design space is large, our proposed pruning method can effectively prune the design space since some directives are conflicting and some are obviously non-optimal (discussed in Section 3.3.1). Moreover, the sampling-based method can handle the design space efficiently. The design space is specified by the users via configuration files, while also considering the characteristics of FPGA designs. For example, the sizes of memory blocks are the power of two. Further, considering the required computation workloads in the applications, the numbers of candidate configurations of directives are limited. If in some circumstances the design space is extremely large, we can use sampling-based methods [134, 135] to help solve the problem.

Table 3.1 Average Runtime of The FPGA Tool for Each Design

Benchmark	HLS (s)	Syn (s)	Impl (s)
`GEMM`	137.75	1379.67	1744.61
`iSmart2`	2206.06	2895.15	4022.38
`SORT_RADIX`	166.66	917.01	1206.94
`SPMV_ELLPACK`	365.91	1528.81	1034.79
`SPMV_CRS`	1333.95	3614.51	1503.30
`STENCIL3D`	369.10	2082.25	969.29
Average	763.24	2069.57	1746.89

3.6 Experimental Results

In our experiments, the initial design space is defined by specifying all of the possible locations of directives and their factors in YAML files. We parse the YAML files and convert the directives to feature vectors and HLS TCL files. The target FPGA board is Xilinx Virtex-7 VC707. The FPGA design tool is Xilinx Vivado 2018.2. Our correlated GP version "CGP" [117] is implemented based on [15] and [105]. The deep version "DCGP" is implemented based on BoTorch [11] and GPyTorch [43].

3.6.1 Integration of Optimization and Deployment

Here we recap the integration of the proposed Bayesian-based optimization techniques and the Xilinx-based FPGA deployment flow, as shown in Fig. 3.11. The Xilinx-based FPGA design tool (*i.e.*, Vivado) is used in both the initialization and the iterative optimization steps to obtain the performance reports. The

HLS optimization directives sampled by our algorithm and the HLS C/C++ sources codes are the inputs to the FPGA design tool. We run the FPGA design tool for the input designs to certain stages with the best acquisition values, according to the acquisition functions in Equation (3.15).

Some HLS directive designs are initially sampled and synthesized with the FPGA tool to build the models. Then, our algorithm iteratively samples HLS directive designs from the design space and then synthesizes them with the FPGA tool. The new designs and performance reports are utilized to update the models. This process stops while convergence, *i.e.*, the sampled designs satisfy the performance requirements.

3.6.2 Objective Selection

Power, performance, and area (PPA) are three popular metrics. To measure the system performance, we choose to use delay (task time length), *i.e.*, the product of latency and clock period. Latency reflects how many clock cycles are needed to finish one task. The clock period is the time length of each cycle, which reflects the congestion information of the designs and is a key design indicator for some applications. We use the utilization of the look-up table (LUT) as the area (*i.e.*, resource consumption) metric. LUT can be used to implement the control logic and simple computations. For tasks requiring high parallelism designs, the LUT utilization is usually the key metric. Other resource metrics (RAMs, DSPs, FFs) can also be easily integrated into the multiple objectives in the same manner. Power as a metric is directly used in the experiments. Compared to works that consider only one or two metrics or linear combinations of these metrics, our work is more practical and challenging.

3.6.3 Benchmarks and Methods

We conduct experiments on six benchmarks. Five are from the open-source FPGA application benchmark MachSuite [98], *i.e.*, `GEMM`, `SORT_RADIX`, `SPMV_ELLPACK`, `SPMV_CRS`, and `STENCIL3D`. Another benchmark is `iSmart2` [60], an object detection deep neural network model deployed on FPGA. `GEMM` is the general matrix multiplication which is widely used to implement the convolutional operators, fully connected operations, and *etc.* Both `SPMV_ELLPACK` and `SPMV_CRS` are for the sparse matrix-vector multiplication while their storage formats are different (`ELLPACK` format and `CRS` format). These two sparse computations are needed by the sparse neural networks. `iSmart2` stacks 12 convolutional modules, including group convolutions, point-wise convolutions, ReLU, pooling layers, and *etc.* `Stencil3D` is the stencil operation used in numerical analysis. We consider the unrolling and pipelining for the loops and partitioning for the arrays. BRAM core is used as the storage resource. DSP48 and Mul_LUT are used as the computation resource cores. The numbers of the consumed computation cores are influenced by

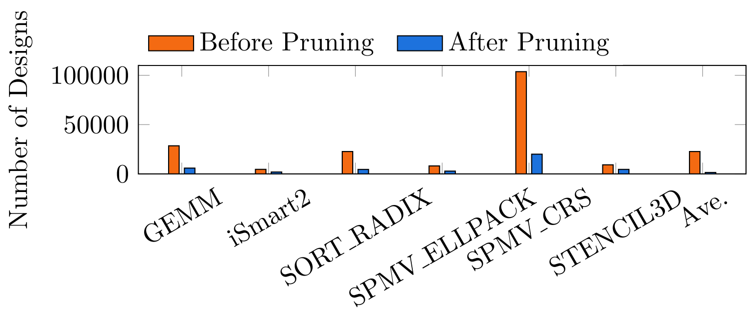


Figure 3.13 Sizes of The Design Spaces Before and After Pruning.

loop unrolling. The storage volumes are influenced by array partitioning. Achieving the optimal allocation solution is non-trivial, while different resource allocations will affect other optimization targets significantly. There are no particular directives that can uniquely determine the allocations, thus making this problem challenging. Our method uses the correlated multi-objective method and deep kernel functions to help find the optimal configurations. Loop tiling cannot be easily handled by the HLS directives and requires more techniques provided by the compilation toolchain (backend code generator). Each benchmark contains a large number of possible configurations. The average runtimes of the FPGA tool for these designs are shown in TABLE 3.1. The sizes of the design spaces before and after the pruning method are plotted in Fig. 3.13.

In the deep kernel function of DCGP, there are four fully connected layers, with output dimensions 1000, 500, 50, and 6. Each of the first three fully connected layers is appended with a ReLU layer. The outputs of the deep kernel functions are the inputs of the GP models. The input features are enlarged into a high-dimension space (*i.e.*, 1000, and 500) by the deep model to learn more information. The dimension is very large compared with the original feature configurations, and that is why this is called the deep method. Then the features are embedded to learn key information with smaller dimensions (*i.e.*, 50, and 6).

Four popular and representative methods are compared with our methods. [83], shorted as FPL18, is also based on Bayesian methods and the Gaussian process. The authors build linear multi-fidelity and independent multi-objective models. [78], abbreviated as DAC19, defines several regression models to guide the FPGA HLS designs with existing ASIC designs. Although the starting points are different, their methods are transferable. *Post-HLS* reports in our problem can be regarded as the ASIC implementations, to predict the *post-Implementation* reports. Randomized Transduction Experimental Design (RTED) proposed by [77] is also used in DAC19 [78] to select better and representative initialization configurations. Artificial neural network (ANN) and Boosting tree (BT) methods have been used in [33, 121, 151] to guide the back-end designs and achieve good performances. For these regression algorithms, some configurations are randomly sampled from the design space to initialize these models. We use the *post-Implementation* reports as the regression targets. For each objective, we build one model. After all of the models are trained, the whole design space is fed into these models to predict the Pareto points. For these learned

Pareto points, we run the Xilinx Vivado design flow to get their real reports. Note that these models are only used to learn the relative numerical relationships to determine the Pareto points. Besides, the various design objectives have performance values that vary greatly in order of magnitude. To guarantee the stability and robustness of the trained models, the performance values are divided by estimated values to scale different target values to a suitable range. For fairness, all of these algorithms use the same feature encodings and design spaces as our method.

Different design objectives have distinctive ranges of performance values, which would mislead the models significantly. For example, the latencies are several seconds while the consumptions of LUTs are hundreds of thousands. The objective values should be normalized during optimizations to avoid inappropriate data shifts. Considering that the maximum power and hardware resources are specified for a known FPGA platform, in practice, we can use these specifications to help adjust the data magnitude. For delay, we estimate a scale factor according to the delays obtained in the initialization stages ($2\times$ of the maximum in the initialization stages). Then the delays are divided by the scale factor before feeding into the model.

3.6.4 Experimental Settings

For our methods and FPL18 [83], we run 10 tests on each benchmark and the results reported in the results are the averages.

For each benchmark, 8 configurations are randomly sampled to initialize the models. The maximum optimization step is 40 and the early stopping factor is 5. There exists a trade-off between the optimization costs and the quality of results. The experimental results show that our methods converge after 30 steps. Therefore, we choose to use 40 steps.

For ANN, we design a model with 2 hidden layers. We train the model with $\{500, 1000, \ldots, 5000\}$ times. For the Boosting method [33, 121, 151], we run a group of experiments, with tree depth from 1 to 6, and learning rate in $\{0.1, 0.2, 0.3, 0.4, 0.5\}$. In DAC19 [78], different numbers of initial sets are sampled to build the models. Therefore, the number of initial sets is also considered as a hyper-parameter, *i.e.*, $\{3, 4, \ldots, 11\}$. In experiments of ANN, Boosting, and DAC19, for each benchmark, the number of initialization configurations is 48. It is worth mentioning that some optimization techniques require initialization and iterative optimization (*e.g.*, ours and FPL18), while some methods have only initialization steps (*e.g.*, ANN). In practical applications, with or without initializations and iterative optimizations is not an issue since the target is to find the optimal solutions. They both run the FPGA design tools to get the actual performance values. For each benchmark, we optimize the configurations from scratch with no prior data. Therefore, the fair comparisons should consider the overall costs of the initializations and the iterative optimizations for different optimization techniques. This kind of cost measures the overall overhead for the users to achieve final optimization results. The overall costs are compared in Section 3.6.7. Although the maximum optimization step is 40, the optimization step is usually fewer than 40 because of the early stopping mechanism (*i.e.*, convergence).

Two metrics are used to measure the performance: average distance to reference set (ADRS) and overall runtime. ADRS computes the distance between the learned Pareto set and the real Pareto set [78].

$$\mathrm{ADRS}(\Gamma, \Omega) = \frac{1}{|\Gamma|} \sum_{\gamma \in \Gamma} \min_{\omega \in \Omega} f(\gamma, \omega), \tag{3.20}$$

where Ω is the learned Pareto set, Γ is the real Pareto set, $f(\gamma, \omega)$ is the distance between two points, $\gamma \in \Gamma$ and $\omega \in \Omega$, $|\Gamma|$ is the number of points in Γ. Overall runtime is the total time needed to get all results, including initialization and iterative optimization. To validate the effectiveness of our method, all the configurations in the design space are run to obtain the whole objective space, though consuming lots of time, huge amounts of computation, and storage resources.

3.6.5 Results and Analysis

As mentioned above, all of the parameters are trained jointly with a unified supervised objective based on the chain rule, as shown in Equation (3.19). An example of the training loss and parameters of our deep kernel function and GP model is shown in Fig. 3.14. The results show that the parameters of the deep

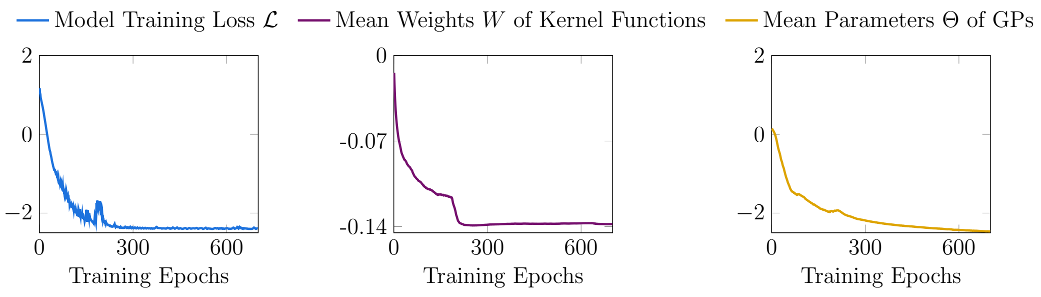


Figure 3.14 The training loss of our method, and the mean parameters (W and Θ) of the deep kernel functions and the GP models. The results show the great convergence of the training loss. The mean parameters W and Θ follow the same convergence trend which validates the effectiveness of our training method.

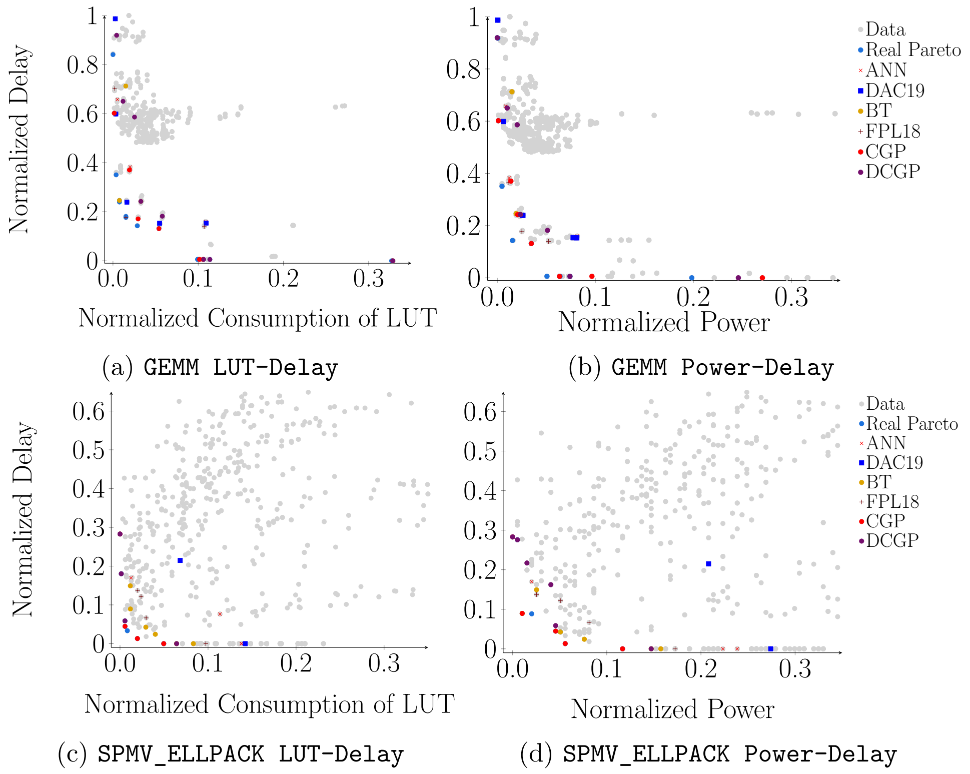
Table 3.2 Normalized Experimental Results

Benchmark	Normalized ADRS						Normalized Standard Deviation of ADRS					
	FPL18	ANN	BT	DAC19	CGP	**DCGP**	FPL18	ANN	BT	DAC19	CGP	**DCGP**
`GEMM`	0.50	1.00	0.65	1.08	0.27	**0.25**	0.46	1.00	0.37	0.90	**0.12**	0.19
`iSmart2`	0.68	1.00	1.28	1.49	0.65	**0.59**	0.75	1.00	1.10	1.24	**0.20**	0.26
`SORT_RADIX`	0.72	1.00	1.09	0.94	0.64	**0.59**	0.57	1.00	1.72	2.28	0.48	**0.27**
`SPMV_ELLPACK`	0.47	1.00	0.22	1.21	0.19	**0.11**	0.24	1.00	0.06	0.99	0.09	**0.01**
`SPMV_CRS`	0.29	1.00	2.09	1.15	0.22	**0.20**	0.26	1.00	2.09	1.52	**0.03**	0.20
`STENCIL3D`	0.41	1.00	0.40	0.41	0.39	**0.31**	0.57	1.00	**0.00**	0.05	0.03	0.05
Average	0.51	1.00	0.96	1.05	0.39	**0.34**	0.47	1.00	0.89	1.16	0.16	**0.16**

kernel function and the GP model have good convergence trends, which validates the performance of our joint training method.

Two examples are plotted in Fig. 3.15 to show the learned Pareto points. For easy visualizations, three objectives are plotted in two figures. The results demonstrate that our learned Pareto points are much more closer to the real Pareto points. All of the statistical results are listed in TABLE 4.1, while expressed as ratios to the results of ANN.

As shown in TABLE 4.1, our methods, CGP [117] and DCGP



(a) `GEMM LUT-Delay` (b) `GEMM Power-Delay`

(c) `SPMV_ELLPACK LUT-Delay` (d) `SPMV_ELLPACK Power-Delay`

Figure 3.15 Learned Pareto designs of `GEMM` and `SPMV_ELLPACK` in the objective spaces.

outperform all of these baselines by a lot. Firstly, compared with FPL18 [83], our methods can achieve much better results on all benchmarks. That is because we consider practical non-linear and correlated relationships in real applications. Secondly, the other three methods are also worse than ours, because they cannot handle complex relationships between multiple fidelities. Thirdly, for benchmarks with complicated code structures, the models without GP models are inferior. For example, the irregular memory accesses of `SORT_RADIX` bring great challenges to ANN, Boosting tree, and DAC19. The results prove that

Table 3.3 Profiling Information of Runtime Details (Optimization Costs, hours)

Benchmark	FPL18		ANN		BT		DAC19		CGP		DCGP	
	Alg.	FPGA	Alg.	FPGA	Alg.	FPGA	Alg.	FPGA	Alg.	FPGA	Alg.	FPGA
`GEMM`	0.46	24.92	0.05	30.03	0.21	30.03	0.09	210.21	0.54	20.42	0.63	18.48
`iSmart2`	0.41	127.83	0.12	145.26	0.39	145.26	0.22	1016.82	0.56	61.01	0.58	59.07
`SORT_RADIX`	1.02	15.19	0.98	32.32	0.98	32.32	1.02	226.24	1.17	10.99	1.06	10.47
`SPMV_ELLPACK`	0.37	16.28	0.03	38.77	0.27	38.77	0.05	271.39	0.48	25.20	0.72	19.25
`SPMV_CRS`	0.48	77.18	0.02	85.75	0.22	85.75	0.06	600.25	0.58	61.74	0.73	60.32
`STENCIL3D`	0.54	18.59	0.07	45.34	0.43	45.34	0.12	317.38	0.63	19.95	0.42	17.44
Average	0.55	46.67	0.21	62.91	0.42	62.91	0.26	440.38	0.66	33.22	0.69	29.39

Table 3.4 Profiling Information of Overall Runtime (hours)

Benchmark	FPL18	ANN	BT	DAC19	CGP	DCGP
`GEMM`	25.38	30.08	30.24	210.30	20.96	**19.11**
`iSmart2`	128.24	145.38	145.65	1017.04	61.57	**59.65**
`SORT_RADIX`	16.21	33.30	33.30	227.26	12.16	**11.53**
`SPMV_ELLPACK`	16.65	38.80	39.04	271.44	25.68	**19.97**
`SPMV_CRS`	77.66	85.77	85.97	600.31	62.32	**61.05**
`STENCIL3D`	19.13	45.41	45.77	317.50	20.58	**17.86**
Average	47.21	63.12	63.33	440.64	33.88	**30.08**

our methods are general enough to handle various applications. Our methods also achieve much better stability according to the standard deviations of ADRSs, as shown in TABLE 4.1. Besides, our deep version DCGP outperforms CGP [117] on all of the six benchmarks with respect to the Pareto results with lower ADRS values, and the same average standard derivations. These results effectively prove the performance of the deep method proposed in Section 3.4.4.

The averages of runtime are listed in TABLE 3.3 and TABLE 3.4 to show that our methods can also save time. For fair, the FPGA times are computed according to the average times listed in TABLE 3.1 and the number of interactions with the FPGA tool. For DAC19, the size of one training data set equals ANN. But it has $3 \sim 11$ training sets. Therefore the average runtime of FPGA tool is 7 times (*i.e.*, $(3+11)/2 = 7$) greater than ANN and Boosting tree. DCGP also consumes less time than CGP [117], thanks to the fewer interactions with FPGA tools. Though DCGP is more complicated compared with CGP, the costs are acceptable with several minutes longer "Alg." times. More analyses are provided in Section 3.6.7.

In summary, our DCGP outperforms our CGP [117] and the other baselines. Numerically, the DCGP wins CGP 12.8% on average ADRS and up to 42.1% on `SPMV_ELLPACK`, thanks to the outstanding performance of using deep kernel learning functions and the joint training method.

3.6.6 Ablation Studies on Acquisition Function

In this section, we compare our acquisition function with max-value entropy search (MES), another popular acquisition function for multi-objective optimization problems. In literature [13,56], the objectives $f^1(\boldsymbol{x}), f^2(\boldsymbol{x}), \cdots, f^M(\boldsymbol{x})$ are modeled using M independent GP models with zero mean and *i.i.d.* noise. Researchers propose maximizing the information gains with respect to the Pareto designs learned in the previous optimization steps to overcome the challenges of computing the acquisition function based on input space's entropy. The GP priors are approximated as $\tilde{f}^i$ and sampled from the M independent GP models [13, 56, 57, 126]. We implement MES and embed it into our optimization framework in place of our correlated acquisition function. The experimental results are listed in TABLE 3.5, under the same experimental settings as our DCGP. The results show that using MES degrades performance. The results prove the outstanding performance of our framework with correlated multi-objective optimization methods.

Table 3.5 Comparisons of Normalized ADRS with MES

Benchmark	DCGP	MES
`GEMM`	**0.25**	0.47
`iSmart2`	**0.59**	0.61
`SORT_RADIX`	**0.59**	0.66
`SPMV_ELLPACK`	**0.11**	0.30
`SPMV_CRS`	**0.20**	0.26
`STENCIL3D`	**0.31**	0.40
Average	**0.34**	0.45

3.6.7 Ablation Studies on Runtime

The DCGP method accelerates the optimization process significantly compared with baselines because it requires fewer optimization iterations. Though more time is needed to train the model in each optimization iteration, the training workload is tiny and can be finished in minutes. The time costs reduced by interacting fewer with the FPGA design flow are exciting. The details are listed in TABLE 3.3 and TABLE 3.4. "Alg."

denotes the time costs of running the optimization algorithms, and “FPGA” represents the time costs to run the FPGA design tool to get the actual performance. The results show that using DCGP reduces the time costs remarkably compared with the baselines.

3.7 Summarization

In this chapter, we solve the problem of FPGA HLS directives design optimization. A pruning method is proposed to prune the design space. Correlated multi-objective multi-fidelity Gaussian process models (CGP) can handle the strong nonlinearities among the multiple fidelities. To the best of our knowledge, the correlated multi-fidelity model is introduced into the HLS directive optimization domain for the first time and has been proven to be effective. The advanced deep version further improves the qualities of the learned Pareto points with shorter running times, by using deep neural networks to enhance the kernel functions. The public benchmarks, *e.g.*, `GEMM` and `SPMV`, and an objective detection deep neural network `iSmart2` are tested. The results prove the outstanding performance of our method.

□ **End of chapter.**

Chapter 4

Deployment via Deep Gaussian Transfer Learning

4.1 Introduction

In this section, we focus on the optimization of deployment configurations. Configurations represent the resource allocations, scheduling, binding of DNN models on hardware platforms, and *etc.* Traditionally, these optimization methods are tightly coupled with hardware architectures and model structures [116, 128, 147]. These methods usually propose some analytical formulations to model the latencies, and characterize the target DNN models and hardware platforms with respect to some properties, *e.g.*, the sizes of layers, and the capacities of buffers. Therefore, these methods cannot be flexibly adapted to different models. Further, some general deployment frameworks are developed, *e.g.*, Halide [72] and TVM [26], which use some auto-tuning algorithms to automatically find the optimal deployment configuration for any given model and hardware platform. For example, XGBoost [23] is used to build a boosted decision tree to predict the deployment performance of configurations. Simulated annealing (SA) is used as the solution searching algorithm. AutoTVM [28], which integrates the above algorithms, is an automatic optimization framework in TVM [26] and achieves outstanding performance. GGA [89] takes advantage of a guided genetic algorithm (GGA) to explore the candidate configurations, under the guidance of an artificially designed scoring calculator. CHAMELEON [7] proposes to use a proximal policy reinforcement learning algorithm to learn the actions to search the configuration space progressively.

However, these automatic frameworks are still unsatisfying. Firstly, the optimization process is slow, resulting from the large configuration space and the time-consuming compilation process. Usually, the configuration space contains more than millions of configurations, *e.g.*, more than 200 million in the first layer of VGG-16. It is inevitable to traverse lots of configurations to guarantee the performance of the searching algorithm. It also takes a long time to compile a configuration and do the inference to get the real on-board latency. Therefore, the overall optimization process is very slow, *e.g.*, longer than a whole day. Secondly, although lots of efforts are required to optimize the deployments of various DNN models, explicit empirics have seldom been drawn from the historical data. Despite that many duplicated works have been done to deploy some models, we usually start from scratch to optimize new models even though they are very similar to what has been deployed before. It is believed that with prior empirics, we can further improve the inference performance. CHAMELEON [7] leverages reinforcement learning to learn the evolution rules of the deployment configurations from the historical data. However, the experimental results show that the performance improvements mainly rely on adaptive sampling (AS) which adjusts the searching scope adaptively, instead of the policies of reinforcement learning. The guided genetic algorithm (GGA) [89] explores the candidate configurations to evaluate the similarities between the new layers and the history data, so as to guide the genetic evolution process. However, this method relies on complex and tricky evolution rules and the high-quality scoring calculator of the similarities. These disadvantages make it hard to be popularized in practical scenarios. And its deployment configurations have worse inference latencies compared with CHAMELEON [7]. Meanwhile, engineers are looking forward to the advent of automatic optimization flows without human interference. Ideally, the inputs of an automatic optimization flow are the historical tuning data with no need for manually designed rules.

To counteract these problems, on the one hand, it is urgent to find an accurate method to estimate the performance quickly without interacting with hardware to compile the model and do the inference. On the other hand, historical information should be fully utilized to guide the deployments of new models. In this section, we propose a novel automatic optimization framework based on deep Gaussian transfer learning. Firstly, a deep Gaussian process (DGP) model is built on the historical optimization data to learn the hidden knowledge related to model structures, hardware characteristics, optimal deployment strategies, and *etc.* Stochastic variational inference is adopted to optimize the DGP. Secondly, when deploying a new DNN model, some efficient initial configurations of this new model are sampled under the guidance of the prior knowledge in the pre-trained DGP model. Maximum-a-posteriori (MAP) estimation is applied to tune the DGP model according to these initial configurations, to make the DGP model accommodate for the new task with no loss of the hidden knowledge. Finally, the tuned DGP model is used as a replacement to the time-consuming compilations and on-board inferences during optimization, to predict the performance values of new configurations accurately. Our tuned DGP model accelerates the optimization process remarkably while reducing the inference latency of the final model deployment simultaneously. We test our method on various types of convolutional layers and networks and results show that our method outperforms the state-of-the-art baselines significantly.

4.2 Transfer Learning Based on Deep Gaussian Processes

To avoid confusion, in the following, the *model* refers to our proposed DGP model, and the DNN model to be deployed on hardware is named as a *task*.

4.2.1 Motivations

In our problem, there are some great challenges, including the undersized available dataset resulting from the time-consuming design flow, and the uncertainties with respect to the characteristics of hardware and models which are hard to measure. Deep learning methods achieve outstanding results on many regression problems, but they are prone to be overfitted with a lack of large training dataset in our problem and be overconfident. Previous researches have shown that as the width of a one-hidden-layer neural network increases to infinity, the network converges to a Gaussian process (GP) model [34, 64, 69], which is a powerful nonparametric distribution and has wide applications [9, 83, 85]. GP method grows in complexity to suit the data and is robust enough to the overfitting on small datasets while providing reasonable predictions as well as uncertainty estimations. However, the GP models are limited by the expressiveness of kernel functions. Learning on a large and richly parameterized space of kernels is expensive, and approximations are at risk of overfitting [19, 39]. A deep Gaussian process (DGP) is a hierarchical composition of GPs that can overcome the limitations of GPs while retaining the advantages [100]. It can be regarded as a multi-layer neural network with multiple, infinitely wide hidden layers [17]. The mapping between layers is parameterized by a GP, and consequently, DGP can provide powerful uncertainty estimations. It performs input warping or dimensionality compression or expansion and automatically learns to construct a kernel that works well on the data. With these advantages, the DGP model is adopted in this chapter, as the performance estimator in the optimization process of deployment configurations.

Considering the diversities and relationships between deployment tasks, it is imperative to transfer the knowledge learned in the source domain (historical task) to the target domain (new task). Some kernel learning methods are used as transfer learning approaches to learn scalable, expressive, and flexible kernels [62, 93, 132]. These methods rely on retraining or joint learning on a large number of tuning points of the new tasks. Note that we want to accelerate the searching process, therefore the slow joint learning and data collections are infeasible in our situation. These transfer learning approaches are also highly coupled with their regression or classification methods, which hinders flexibility. Traditionally, Gaussian process models are fitted from scratch via maximum likelihood estimation. In this chapter, we propose a novel transfer learning algorithm based on the maximum-a-posteriori (MAP) estimation. The knowledge learned on history is used as the prior of DGP. Then DGP is tuned via MAP. This has similar philosophies with [84, 125], which also introduce a prior in the model and then calibrate it via posterior. Thanks to the knowledge learned on history, our method does not require much data on the new task. MAP is also easy to be solved with low workloads, so as to accelerate the search of optimums and improve the quality simultaneously. With these advantages, MAP has been widely used recently, *e.g.*, reinforcement learning [110], structured prediction [46], and statistical inference [53].

4.2.2 Our Automatic Optimization Framework

The overall optimization framework is shown in Fig. 4.1. The DNN tasks are optimized layer by layer, following previous arts [7, 26, 89]. Before starting to optimize a new task, a deep Gaussian process model is built on the historical optimization data. The DGP preparation step is task-independent, *i.e.*, the pre-trained DGP model can be used to deploy other tasks. In Fig. 4.1, the DNN task is represented as a graph, in which each node is a layer. For each layer, a searching space $\mathcal{D}$ containing all of the configurations of this layer is generated. In the tuning stage, the pre-trained DGP model is utilized as the criterion to sample some efficient initial configurations from $\mathcal{D}$. These initial configurations are then compiled and deployed to get their on-board performance values. These configurations and performance values are denoted as a tuning set. The hyper-parameters of the pre-trained DGP model are introduced as the prior. Maximum-a-posteriori (MAP) estimation is used to tune the pre-trained DGP model under the guidance of the tuning set. The tuned DGP model is then adopted as the performance estimator in the third stage (*i.e.*, the optimum searching stage). Various algorithms can be applied here as the searching algorithm to find optimal configurations. In experiments, we use simulated annealing as the searching algorithm. The previous arts [7, 26, 89] interact with hardware iteratively in the searching process to obtain the real on-board performance. By contrast, our tuned DGP can take the place of the real hardware and report the predicted performance, so as to accelerate the searching remarkably. All of the configurations found by the searching algorithm are recorded and the final optimal deployment configuration for this layer is selected from the record. The pseudo-code of our framework is provided in the appendix.

4.2.3 Deep Gaussian Processes with Stochastic Variational Inference

Denote our task as $f : \boldsymbol{x} \rightarrow y$, with the deployment configuration vector $\boldsymbol{x}$ and its performance value y. The historical optimization record is $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{y}\}$, with $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ and $\boldsymbol{y} = \{y_1, \ldots, y_N\}$. For a single layer Gaussian process, the non-parametric Gaussian process places a GP prior over the value

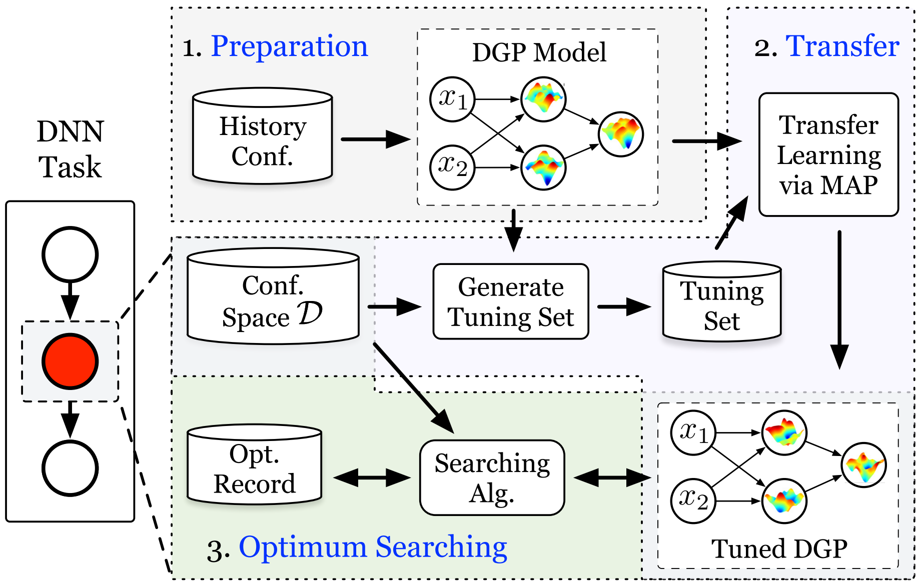


Figure 4.1 Our automatic optimization framework, consists of three stages, *i.e.*, stage 1: DGP model preparation based on the history data, stage 2: transfer knowledge to new DNN layers, and stage 3: optimal configuration searching.

function f as $f(\boldsymbol{x}) \sim \mathcal{GP}(\mu(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x}'))$, where $\mu(\cdot)$ is the mean function and $k(\boldsymbol{x}, \boldsymbol{x}')$ is the kernel function. $\boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X})$ denotes the kernel matrix, *i.e.*, $\boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X})_{i,j} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. Given the historical data $\mathcal{D}$, $\boldsymbol{y}$ is assumed to be influenced by the zero-mean Gaussian noise $\epsilon \sim \mathcal{N}(0, \sigma_e^2)$, *i.e.*, $y_i = f(\boldsymbol{x}_i) + \epsilon$. The noise is indispensable to characterize the hardware uncertainties which may be caused by real-time workloads, temperature fluctuation, and *etc*. The function values with respect to $\boldsymbol{X}$ are denoted as a vector $\boldsymbol{f}$. Denote the hyper-parameters as $\boldsymbol{\theta}$, including noises and parameters in the kernel function. The marginal likelihood takes the form shown in Equation (4.1).

$$\mathcal{P}(\boldsymbol{y}|\boldsymbol{\theta}) = \prod_{i=1}^{N} \int p(y_i|f_i)p(f_i)\mathrm{d}f_i = \mathcal{N}(\boldsymbol{\mu}, \tilde{\boldsymbol{K}}), \tag{4.1}$$

where $\boldsymbol{\mu}$ is the mean vector, $\tilde{\boldsymbol{K}} = \boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X}) + \sigma_e^2 \boldsymbol{I}_N$, and $\boldsymbol{I}_N$ is the identity matrix.

A DGP model stacks multiple single-layer Gaussian processes. The outputs of the previous GP layer are the inputs of the next GP layer. For a DGP model comprised of L layers, denote the value functions of these L layers as $\{f^1, \cdots, f^L\}$. Correspondingly, the function values on inputs $\boldsymbol{X}$ are $\{\boldsymbol{f}^1, \cdots, \boldsymbol{f}^L\}$. Here $\boldsymbol{f}^0$ is defined as $\boldsymbol{X}$. The hyper-parameters in the l-th layer are represented as $\boldsymbol{\theta}^l$. Based on the definitions of the single-layer Gaussian process, the prior of a DGP model comprising L layers can be written as Equation (4.2).

$$\begin{aligned} &\mathcal{P}(\boldsymbol{f}^l) = \mathcal{N}(\boldsymbol{\mu}^l, \boldsymbol{K}^l),\ l = 1, \cdots, L, \\ &\mathcal{P}(\boldsymbol{y}, \{\boldsymbol{f}^1, \cdots, \boldsymbol{f}^L\}) = \prod_{i=1}^{N} \mathcal{P}(y_i|\boldsymbol{f}_i^L; \boldsymbol{\theta}^L) \prod_{l=1}^{L} \mathcal{P}(\boldsymbol{f}^l|\boldsymbol{f}^{l-1}; \boldsymbol{\theta}^l), \end{aligned} \tag{4.2}$$

where the hyper-parameters $\boldsymbol{\theta}^l$ are solved via maximum likelihood estimation which is computationally expensive. For the training set with N configurations, the computation complexity of the gradients of $\tilde{\boldsymbol{K}}$ with respect to $\boldsymbol{\theta}^l$ is $\mathcal{O}(N^3)$. Besides, the marginal density is unavailable in closed form or requires exponential time to compute [14], thus making the inference hard.

Inspired by recent works on posterior approximations of sparse Gaussian approximations, the stochastic variational inference

[14] is employed to accelerate the computations of DGPs in our method. The key technique is to introduce an inducing configuration set $\boldsymbol{Z} = \{\boldsymbol{z}^1, \cdots, \boldsymbol{z}^L\}$ with $|\boldsymbol{z}^l| \ll N$ and $\boldsymbol{z}^1 \subset \boldsymbol{X}$. Denote the function values at configurations $\boldsymbol{z}^l$ as $\boldsymbol{u}^l$ in the l-th layer. The basic assumption is that $\{\boldsymbol{u}^l\}_{l=1}^L$ is a sufficient statistic for $\{\boldsymbol{f}^l\}_{l=1}^L$, so that the real posterior $\mathcal{P}(\{\boldsymbol{f}^l, \boldsymbol{u}^l; \boldsymbol{\theta}^l\}_{l=1}^L | \boldsymbol{y})$ can be approximated given a Gaussian distribution $\mathcal{Q}(\{\boldsymbol{u}^l\}_{l=1}^L)$. To achieve the best $\{\boldsymbol{u}^l\}_{l=1}^L$, we minimize the KL divergence between $\mathcal{P}(\{\boldsymbol{f}^l, \boldsymbol{u}^l; \boldsymbol{\theta}^l\}_{l=1}^L | \boldsymbol{y})$ and $\mathcal{Q}(\{\boldsymbol{f}^l, \boldsymbol{u}^l\}_{l=1}^L)$ with respect to the selection of $\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L$, as shown in Formulation (4.3).

$$\min_{\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L} \mathcal{KL}\left(\mathcal{Q}(\{\boldsymbol{f}^l, \boldsymbol{u}^l\}_{l=1}^L) \| \mathcal{P}(\{\boldsymbol{f}^l, \boldsymbol{u}^l; \boldsymbol{\theta}^l\}_{l=1}^L | \boldsymbol{y})\right). \tag{4.3}$$

Formulation (4.3) can be transferred to be the equivalent formulation as follows:

$$\begin{aligned} &\max_{\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L} \log \mathcal{P}(\boldsymbol{y} | \{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L) \\ = &\max_{\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L} \left[\sum_{i=1}^N \mathbb{E}_{\mathcal{Q}(f_i^L | \boldsymbol{u}; \boldsymbol{x}_i, \boldsymbol{Z})}[\log \mathcal{P}(y_i | f_i^L; \boldsymbol{\theta}^L)]\right] \\ &- \sum_{l=1}^L \mathcal{KL}[\mathcal{Q}(\boldsymbol{u}^l) \| \mathcal{P}(\boldsymbol{u}^l; \boldsymbol{\theta}^l)], \end{aligned} \tag{4.4}$$

where $\mathcal{P}(\boldsymbol{y} | \{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L)$ is the likelihood function. The model hyper-parameters $\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}_{l=1}^L$ are solved via Formulation (4.4). For simplicity, denote $\{\boldsymbol{z}^l, \boldsymbol{\theta}^l\}$ as $\tilde{\boldsymbol{\theta}}^l$. Until now, we have finished the training of our DGP model based on the historical data $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{y}\}$.

In our method, the DGP model together with stochastic variational inference grows in complexity to suit the historical data and is robust enough to provide reasonable errors that would result from hardware or system uncertainties [14, 17, 100]. It also has a greater capacity to generalize and contains more hidden information compared with previous arts. The experimental results show us an outstanding performance by using our DGP and its high transferability.

4.2.4 Transfer Knowledge to New Tasks

To transfer the hidden knowledge and empirics from the known historical tasks (*a.k.a.*, source tasks) to new tasks (*a.k.a.*, target tasks), two steps are required, *i.e.*, finding a good initial tuning data set and choosing a fast and efficient transfer learning algorithm.

Firstly, to guarantee that adequate knowledge is learned for the new task, it is crucial to find a good tuning data set. Randomly picking some initial configurations from the extremely large design space would introduce some illegal configurations (*i.e.*, with performance values equal to zero) which cannot help us but wastes lots of time to compile them. Besides, there is no guarantee that the historical data set is large enough to cover the data distribution of the target task. The target task would also possibly have a higher upper bound of performance values, which means the mean value of the historical data might be smaller than the mean value of the target task. Therefore, the tuning data set should contain configurations with performance values as higher as possible, to calibrate the mean function.

To handle these, the DGP model learned from the historical data is used as the empirical criterion to select suitable initial configurations for the new task. A set which is large enough is randomly sampled from the search space and then fed into the DGP model to get the predicted performance. We sort these initial configurations according to their predicted performance values and the configurations with top s performance values are chosen as the tuning points, *i.e.*, $\boldsymbol{X}^t = \{\boldsymbol{x}_1^t, \cdots, \boldsymbol{x}_s^t\}$. The configurations in $\boldsymbol{X}^t$ are compiled and deployed on real hardware to get the real performance set $\boldsymbol{y}^t = \{y_1^t, \cdots, y_s^t\}$. Denote $\{\boldsymbol{X}^t, \boldsymbol{y}^t\}$ as $\mathcal{D}^t$, and then $\mathcal{D}^t$ is used as the tuning set to calibrate the empirical DGP model. Intuitively, a tuning set with high diversities is better. However, in our context, the configuration space is too large and only small parts have good performance. Our target is to find the optimal configurations instead of characterizing the whole configuration space. In other words, we are interested in a small part of the solution space with high performance. Besides, compared with the large space, the size of the randomly sampled set and the number of the sorted configurations are small, a basic situation is that these sampled configurations will always scatter with high diversities in the space. Therefore, finding a better initial tuning set via our DGP is wise with no harm to the diversities.

Secondly, a fast and efficient transfer learning algorithm based on MAP is proposed to tune the DGP model with $\mathcal{D}^t$. As mentioned above, the widely-used transfer learning algorithms [62, 93, 132] are unsuitable in our situations for several reasons. For the target task, with the help of MAP, the model parameters are optimally determined by combining the hidden knowledge (in the form of the parameters $\tilde{\boldsymbol{\theta}}_l$) and $\mathcal{D}^t$. For the convenience of explanation, we omit the layer indices to lighten the notations. Denote all of the parameters in the source task DGP model as $\tilde{\boldsymbol{\theta}}$ and the parameters for target task as $\hat{\boldsymbol{\theta}}$. According to the Bayes' theorem, MAP is to find the optimal value of $\hat{\boldsymbol{\theta}}$ (*i.e.*, most likely to occur) to maximize the posterior distribution $\mathcal{P}(\hat{\boldsymbol{\theta}}|\boldsymbol{y}^t)$. Specifically, $\mathcal{P}(\hat{\boldsymbol{\theta}}|\boldsymbol{y}^t)$ follows Formulation (4.5).

$$\mathcal{P}(\hat{\boldsymbol{\theta}}|\boldsymbol{y}^t) \propto \mathcal{P}(\hat{\boldsymbol{\theta}}) \cdot \mathcal{P}(\boldsymbol{y}^t|\hat{\boldsymbol{\theta}}), \tag{4.5}$$

where $\mathcal{P}(\boldsymbol{y}^t|\hat{\boldsymbol{\theta}})$ is the likelihood function in Formulation (4.4). The prior of $\hat{\boldsymbol{\theta}}$ is assumed to follow a Gaussian distribution with $\tilde{\boldsymbol{\theta}}$ as the mean value [125]. To accelerate the computation, the MAP is implemented as an L2 regularization term of $\hat{\boldsymbol{\theta}}$ and $\tilde{\boldsymbol{\theta}}$, *i.e.*, $||\hat{\boldsymbol{\theta}} - \tilde{\boldsymbol{\theta}}||_2^2$. The objective function to tune the parameters is defined as Formulation (4.6).

$$\max_{\hat{\boldsymbol{\theta}}} \quad \log \mathcal{P}(\boldsymbol{y}^t|\hat{\boldsymbol{\theta}}) - \lambda||\hat{\boldsymbol{\theta}} - \tilde{\boldsymbol{\theta}}||_2^2, \tag{4.6}$$

where λ is a hyper-parameter. Theoretically, L2 regularization is equivalent to MAP inference with a Gaussian prior on the parameters [48]. Compared with the traditional GP methods which are fitted from scratch, our method with prior $\tilde{\boldsymbol{\theta}}$ does not require too much data and saves time.

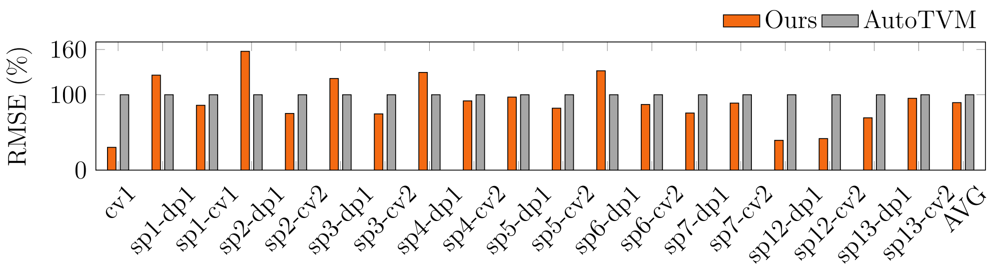
An important characteristic of deployment is that various computation operations would have a non-negligible influence on the communication modes, resource allocations, and *etc.* For example, depthwise convolutions and direct convolutions have distinct computation patterns. These characteristics are hard to be summarized as a unified rule even for senior engineers. To guarantee the performance of our flow, DNN layers are categorized into some groups according to their types. The knowledge is learned and transferred within each group.

4.3 Experiments

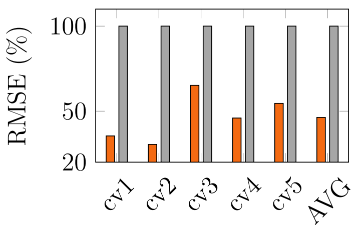
We implement our flow based on GPyTorch [43] and embed it into TVM to validate the performance. Some ablation studies are conducted. Layer-wise and model-wise performance are analyzed and compared with the state-of-the-art. Due to space limitations, we present some important settings and representative results, and more details and results can be referred to the appendix.

4.3.1 Experimental Settings

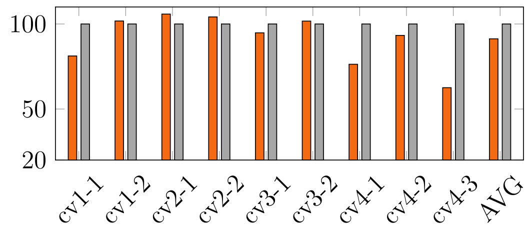
Our experiments are running on an Intel(R) Xeon(R) E5-2680 v4 CPU@ 2.40GHz. The hardware platform is an NVIDIA GeForce GTX 1080Ti GPU and the CUDA version is 9.0.176.



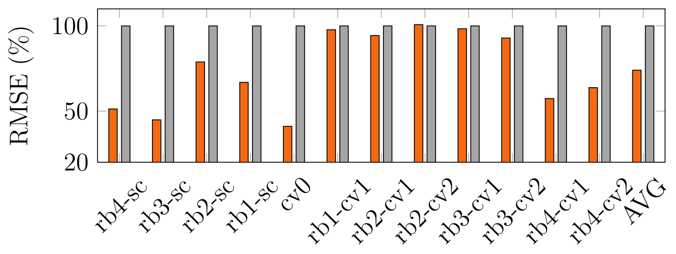
(a) MobileNet-v1



(b) AlexNet



(c) VGG-16



(d) ResNet-18

Figure 4.2 RMSE of our predicted GFLOPS, the data are expressed as the ratios to the results of XGBoost in AutoTVM. Here, our DGP is directly used to predict the GFLOPS of new tasks without tuning. cv: convolution, rb: residual block, sc: shortcut, sp: separable convolution, dp: depthwise convolution.

For fair comparisons, models tested in previous work [7, 28, 89] are tested, including AlexNet [67], ResNet-18 [54], and VGG-16 [108]. Further, MobileNet-v1 [58] is tested. Note that the current deployment flows [7, 28, 89] optimize the DNN models layer by layer. The optimization algorithms and processes are independent of the model structure, therefore simple model structures are enough to validate our method with no need of using more complicated DNN models. The representative DNN layers widely used in both industries and academia are covered in these models, including convolutional layers, residual blocks, depthwise separable convolutional layers, and *etc.* For the layers with the same structures, only the first of them will be optimized and others directly use the same configuration for this layer. Besides, same with [7, 28, 89], we focus on the optimizations of various convolutional layers, and other types of layers are skipped, *e.g.*, fully connected layers and pooling layers. These other layers directly use the settings provided by TVM.

AutoTVM [28], integrating XGBoost, simulated annealing, and *etc.*, is used as the baseline. Besides, two outstanding baselines are also compared, including DAC'20 GGA [89] which uses a well-designed heuristic guided genetic algorithm, and ICLR'20 CHAMELEON [7] which is based on reinforcement learning and adaptive sampling algorithm. In our method, we use the simulated annealing in TVM as the searching algorithm and follow the same settings as AutoTVM and CHAMELEON. Our DGP is used as the performance estimator and a replacement to GPU during configuration searching, as mentioned in Section 4.2.2. The radial basis function is adopted as the kernel function. The number of inducing points in variational inference is 128. Notably, the experimental platforms and software versions are distinct compared with other works, which would have a significant effect on the search time and the performance of the final deployments. For fairness, the results are expressed as ratios to the results of AutoTVM.

To illustrate the performance, three metrics are used, *i.e.*, Giga floating operations per second (GFLOPS), the reduction of the inference latency, and the reduction of the search time to find the optimal configuration. GFLOPS measures the number of floating-point operations conducted by the hardware per second during executing the model. It is used as the optimization objective for each layer in our method and the baselines. Inference latency is the final end-to-end on-board inference time of the whole model. Search time is the overall time cost to optimize the deployment of the whole DNN model, including the interactions with hardware and model tuning.

Note that in the current single GPU stream deep learning implementations, the GPU resources are occupied exclusively by the tasks during inference. Therefore, the onboard memory consumptions, the numbers of blocks and threads, *etc.*, are not compared since we rely on this basic assumption about resource usage.

4.3.2 Layer Groups and Historical Data

As mentioned above, to guarantee the transferability of prior knowledge, the DNN layers are categorized into some groups. We choose three criteria, including layer type (*e.g.*, direct convolution, or depthwise separable convolution), kernel size (*e.g.*, 3×3, or 7×7), and padding type (*e.g.*, no padding, or padding size = 1). These criteria are fundamental factors that would have a great influence on the deployment performance and usually bother engineers. According to these criteria, VGG-16 has 1 group, while ResNet-18, AlexNet, and MobileNet-v1 have 4, 3, and 4 groups, respectively. The first layer of each group is deployed via AutoTVM and the configurations explored by AutoTVM in this process are collected as the historical data for this group. This makes our method more practical in demanding application scenarios.

4.3.3 Ablation Studies on the Proposed DGP

We perform ablation studies on our pre-trained DGP model, *i.e.*, evaluate the results of stage 1 in Fig. 4.1. The accuracies of directly using DGP trained on the historical data to predict the performance of configurations of new layers are plotted, in comparison with the prediction performance of the XGBoost used by AutoTVM. For fair comparisons, these two methods use the same training data, as mentioned in Section 4.3.2. After training, they are directly used to predict the performance without tuning. The root-mean-square error (RMSE) of the predicted GFLOPS values is to characterize the prediction error. The results are shown in Fig. 4.2. The prediction accuracy of our DGP on direct convolutional layers outperforms XGBoost

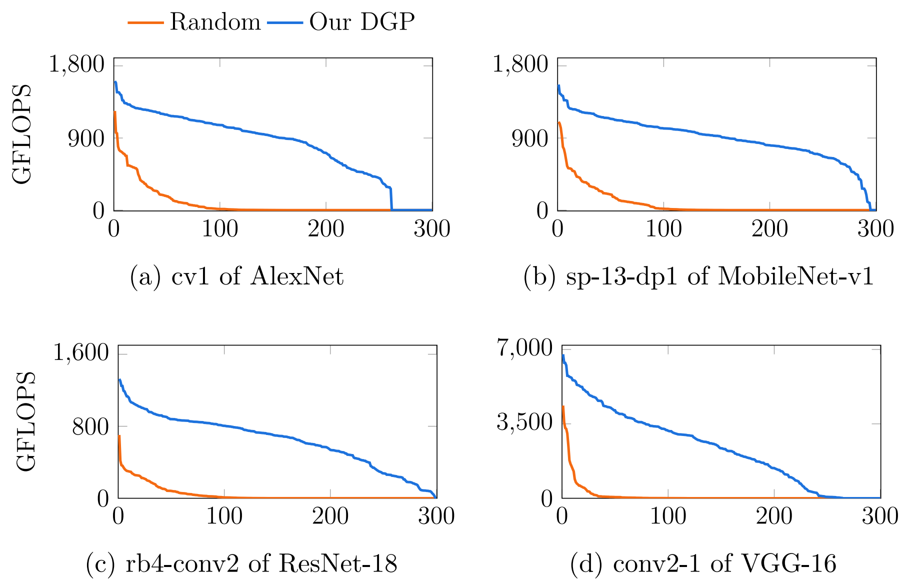
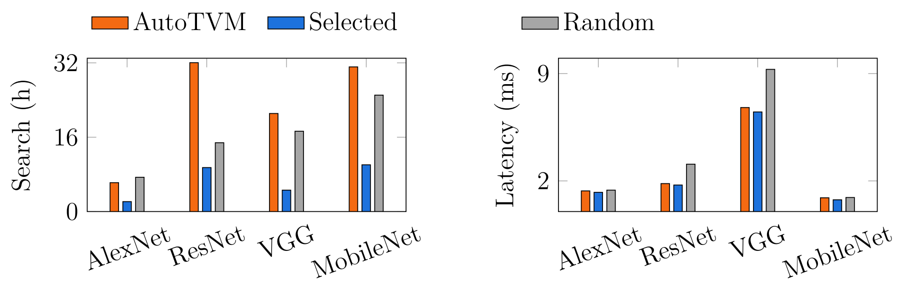


Figure 4.3 The randomly sampled tuning set and the set selected according to DGP. The data are in descending order. There are 300 configurations in each tuning set, and the X-axis is the index of the configuration.

remarkably, no matter whether with padding or not, or with various sizes of kernels, or different sizes of strides. Our DGP wins on most of the depthwise convolutional layers. As to the performance of residual blocks, our method is also the superior one. On these four models, our average results are the best. It is demonstrated that our DGP models are able to learn enough prior knowledge of the hidden characteristics of the hardware architecture, model structures, and *etc.*

As mentioned above, the pre-trained DGP is used as the empirical criterion to select a suitable initial configuration set for the subsequent tuning stage. Note that our target is to learn the good configurations instead of the whole configuration space.



(a) Search Time (b) Inference Latency

Figure 4.4 Comparisons between AutoTVM and ours. “Selected” means the tuning configurations are selected by using our pre-trained DGP as the criterion. “Random” means the tuning configurations are randomly sampled from the configuration space without any prior knowledge.

Using our DGP will help choose the useful configurations and will teach the model to learn more about the characteristic of this layer. Examples of the sampled configurations and their on-board GFLOPS values are plotted in Fig. 4.3. In experiments, the tuning set contains 300 configurations. Most of the configurations sampled via our DGP model are feasible and have continuous GFLOPS values. In comparison, most of the randomly sampled configurations are infeasible on hardware. Besides, the maximum GFLOPS of the random method is lower than ours which means the DGP tuned on the random set is unable to give higher prediction values for good configurations.

4.3.4 Ablation Studies on the Transfer Learning

To prove the effectiveness of our transfer learning method, we compare the results of using the randomly sampled tuning set

Table 4.1 Comparisons of Search Time and End-to-end Model Inference Latency

Model	AutoTVM [28]		ICLR'20 [7]			Ours				
	Search (h)	Infer. (ms)	Search Red. %	Infer. Red. %	HV	Search (h)	Search Red. %	Infer. (ms)	Infer. Red. %	HV
`MobileNet-v1`	31.14	0.8980	-	-	-	10.06	67.69	0.7664	**14.65**	**9.9168**
`AlexNet`	6.28	1.3467	72.16	5.88	4.2409	2.14	65.96	1.2537	**6.91**	**4.5573**
`VGG-16`	19.92	6.7847	82.56	3.44	2.8418	4.61	76.83	6.4972	**4.24**	**3.2556**
`ResNet-18`	32.04	1.8248	76.67	4.16	3.1915	9.47	70.43	1.7305	**5.17**	**3.6423**

with the results of using the tuning set selected by our DGP (as mentioned in Section 4.2.4). The results are shown in Fig. 4.4. The randomly sampled tuning sets increase the inference latencies significantly because the performance of most of the sampled configurations is unsatisfying. Randomly sampling a small number of configurations cannot introduce enough knowledge about the optimal configurations, but confuses the pre-trained DGPs.

4.3.5 Performance of the Whole framework

Some results with respect to the reduction of search time of the whole optimization process and reduction of model inference latency are analyzed here, compared with the state-of-the-art baselines. The detailed results are listed in TABLE 4.1. The reported latency is the average of latencies from 1800 on-board inference trials and is believed to be accurate enough. Usually, there is a trade-off between search time and model inference latency. To improve the model inference performance, more configurations are sampled and analyzed in the searching process.

Consequently, the search time increases, and vice versa. For fair comparisons between these two closely related and interacting metrics, we introduce the concept of hypervolume (HV) [130]. Hypervolume is commonly adopted to measure the solutions of multi-objective optimization problems. The reduction ratios of inference latency and search time are multiplied, to measure the overall performance, as shown in Equation (4.7).

$$\text{HV} = \text{Redu. of Latency} \times \text{Redu. of Search Time} \times 100. \tag{4.7}$$

Here the HV value is multiplied by 100 to adjust the order of magnitude. The solution with a higher HV value is the better one. The results prove the superior performance of our method. Compared with ICLR'20 CHAMELEON [7], though our reductions in search times are not optimal, the reductions of the inference latencies are much better. Our overall results are much better than CHAMELEON, with respect to the HV values. In GGA [89], the authors reduce the search time of ResNet-18 by 93.17% and reduce the inference latency by 3.26%. Although they have the fastest search speed, the inference latency is the worst. The HV value of their method is 3.037, which is also worse than [7] and ours. Accelerating the search speed too aggressively does not worth the loss of the quality of results. The precise search times and inference latencies of VGG-16, AlexNet, and MobileNet-v1 are not provided in GGA [89]. For supplementary, the GFLOPS values of VGG-16 are plotted in Fig. 4.5. Compared with AutoTVM, our method wins on most layers and

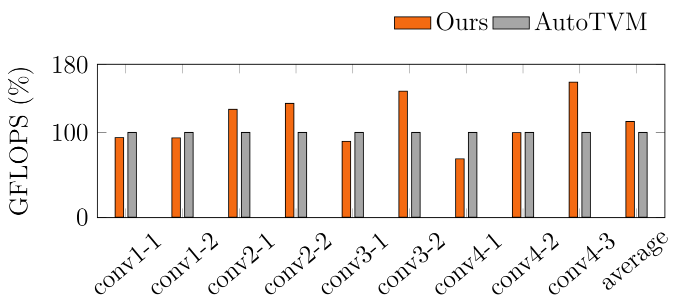


Figure 4.5 The ratios of the GFLOPS values of VGG-16.

has a better average GFLOPS value. As mentioned before, the resources are used exclusively by the deep learning tasks during inference in the single-stream implementations. Therefore, the memory consumptions are not compared in our work and the baselines [7, 89]. Besides, for different GPU devices, the learning-based techniques can always fit the onboard resources to tune the deployments and achieve the best latencies. Reducing memory consumption is not an optimization target anymore.

Despite the existing trade-off between the searching time and the inference latency, on-chip inference latency is actually the most critical metric since the model deployment is “once for all”, which means no matter how much time we spent to optimize the deployment, the faster on-chip inference is more important than the faster optimization process. From this perspective, our method also outperforms the baselines significantly.

4.4 Conclusion

In this chapter, a transfer learning algorithm based on a deep Gaussian process (DGP) is proposed to optimize the deployment of DNN models, by using the historical information efficiently. The representative DNN layers and models are tested. Both the search time and inference latency are reduced simultaneously. The experiments show that our method outperforms the baselines remarkably.

□ **End of chapter.**

Chapter 5

Tuning Computations via Graph Attention Network

5.1 Introduction

The great successes of deep learning algorithms in this AI era [45,54,99,145] stimulate the fast development of high-performance computing for deep neural networks. Many techniques have been developed to optimize the on-chip inference performance, tackle heavy workloads, and bridge the gap between hardware designs and algorithm developments. Some representative arts include algorithm compression and pruning [22,51], hardware/algorithm co-optimization [146], neural architecture search [38], *etc.*

Some compilation frameworks have been proposed to optimize the model inference on GPU. Halide [72] and TVM [26] propose to decouple the model analysis and backend code optimization and use the auto-tuning algorithms to tune the optimal deployment configuration for DNN models. The models are analyzed and partitioned into some small subgraphs. Each subgraph is implemented as a *kernel* on GPU and some parameters in these kernels are tuned to achieve the best performance. Candidate values of these parameters are termed *knobs* [7], or *annotations* [153]. The genetic algorithm (GA) and simulated annealing (SA) are the typical parameter-tuning algorithms.

Based on TVM, many techniques are proposed to help tune the kernel implementations. AutoTVM [28] provides some fixed optimization rules and code templates for the DNN operations on GPUs and encodes the parameters in the templates as fixed-length feature vectors. These features contain the numbers of on-device threads, virtual threads, blocks, *etc.* In the genetic algorithm process, AutoTVM interacts with GPU to collect the on-device performance and trains an XGBoost model as the performance estimator of the feature vectors to guide the search for optimal parameters. The optimization process is slow, and no historical tuning data are utilized. Based on AutoTVM, an advanced active learning method [114] is designed to learn representative parameters in the optimization process. CHAMELEON [7] introduces reinforcement learning to learn the searching strategies from the history tuning data and adapts the searching space during optimization. Guided Genetic Algorithm (GGA) [89] improves AutoTVM by using some heuristic rules to guide the genetic algorithm. The similarities between new deployment tasks and the history tuning data are computed to measure the performance of new knobs. DGP-TL [113] proposes to use deep Gaussian process models to learn the history data and use transfer learning with fine-tuning to guide the new DNN deployment tasks. These methods prove the effectiveness of the learning-based methods. Further, Ansor [153] designs some complicated rules to generate code "*sketches*" for the computational subgraphs in models to break the shackles of fixed templates. The sketches are high-level program structures and leave billions of low-level parameters as "*annotations*". The features representing these codes are *statistical* which are more complicated, including parallelism in multiple programming levels, type of memory access, the number of touched cache lines, the number of floating/integer multiplication-add operations, *etc*. With the advantages of flexible sketches and annotations, Ansor outperforms the AutoTVM-based previous arts significantly.

Despite the advancements, existing frameworks are still unsatisfying. Firstly, the *structural* information of the computational subgraphs is underutilized. Existing techniques rely on the statistical information of codes to train a cost model as the performance estimator [7, 28, 89, 113, 153, 154] while the structural information is discarded. The structures reflect the topological relationships and scheduling information of the operations, which greatly influence the inference performance, while the statistical features fail to characterize the structures. Modern DNN models contain different structures. Learning-based techniques which only rely on statistical information are incapable of measuring these diversities. Secondly, the feature items in the statistical feature vectors are treated equally, despite their physical meanings and relationships related to the implementation details. Each statistical feature vector is usually directly passed to the learning models to predict its performance. Therefore, the complicated but implicit relationships between the feature items are not taken into considerations.

To tackle the above problems, a novel method, GTuner, is built based on TVM and Ansor to tune the computations of deep neural networks on GPU. The structural information of the computational graphs and statistical code features are utilized. The complicated relationships between the features are learned automatically. The contributions are summarized as follows:

- A novel method, GTuner, is proposed with a graph attention network (GAT) as the performance estimator. GAT comprises a graph neural network (GNN) module to aggregate structural information and a multi-head self-attention (MHSA) module to mine inter-feature relationships.

- Structural information of the computational subgraphs is extracted from the intermediate representations of the compilation flow with the help of our code parser and analyzer. Then the information is propagated and aggregated via the graph neural layers to learn high-quality features for the graphs.

- The MHSA module is designed to learn the complicated but implicit relationships between the structural and code statistical features via the self-attention mechanism. The drawbacks of losing structural information and long-range dependencies between the features are overcome.

- With the GAT, GTuner optimizes the kernel codes for GPU efficiently. The results demonstrate the remarkable performance of GTuner compared with the baselines.

5.2 Preliminaries

5.2.1 Computational Graphs

The deep neural network models are usually represented as computational graphs, with layers (operators) as nodes and the edges representing communications in the models and reflecting the topological information. These computational graphs are mapped to the hardware accelerators (*e.g.*, FPGA [52], GPU [111], ASIC [30] and TPU [63]) through some deep learning frameworks and libraries (*e.g.*, cuDNN [31], oneDNN [2], and TensorFlow [6]). The implementation details for the FPGA or ASIC are available for the designers, such as the allocations of the MAC engine, systolic array, cache behavior, and computation patterns [30, 61, 63, 80]. Therefore, accurate models, even analytical formulations, can be built to measure the performance. In contrast, for GPU, the complexities of the hardware and programming model and the lack of implementation details make this problem challenging.

Existing compilation techniques partition the graphs into small subgraphs. Each subgraph contains several neighboring layers, *e.g.*, softmax, pooling, linear layers, and convolutions [26, 153]. Each subgraph is implemented as a kernel on GPU. Many code templates with some parameters to be determined are designed to implement the computations of a kernel. Some parameters are loop boundaries, splits, cache read steps, inlines, *etc.* The final executable kernel codes are generated according to these parameters. As mentioned above, in Ansor, the code templates and parameters are termed *sketches* and *annotations*, respectively. Fig. 5.1 and Fig. 5.2 show an example of one sketch and its two annotations. Sketches for the same subgraph may have distinct structures with various numbers and orders of loops and computations. Implementing a unified encoding method to represent the different sketches and annotations is difficult. Therefore, Ansor extracts statistical features for each code annotation via static code analyses, *e.g.*, number of touched cache lines, touched memory in bytes, number of floating multiplication operations, sizes of allocated buffer in bytes, *etc.* Some important concepts are clarified here:

- **Computational graph and subgraph**: the whole DNN model is represented as a computational graph, and the graph is split into some subgraphs by the DNN compilers.

- **Sketch**: the code templates designed by compilers to implement the computations of a subgraph are termed *sketches*.

```
Generated Kernel Code Sketch:
[Placeholder: A, B
          for i.0 in range(None):
                for j.0 in range(None):
          for ic.2 in range(None):
                for jc.2 in range(None):
                     for k.0 in range(None):
               for k.1 in range(None):
                     for k.2 in range(None):
                      for i.3 in range(None):
                      for j.3 in range(None):
                            C = …  ]
```

Figure 5.1 An example of the sketch.

Each subgraph has many templates, *i.e.*, many sketches. An example of the sketch is shown in Fig. 5.1.

- **Annotation**: the combinations of the parameters to be determined in each sketch are termed *annotations*, *i.e.*, each sketch has many annotations, as shown in Fig. 5.2.
- **Structural feature**: each node in a subgraph has some structural features. The nodes' features are aggregated as the structural features of the subgraph.
- **Statistical feature**: each annotation of the subgraph has a statistical feature vector reported by Ansor via static program analyses.

5.2.2 Graph Neural Networks and Attention Mechanism

Graph neural networks (GNNs) have been widely used in modeling graph data, achieving impressive results in the prediction,

```
Annotation 1:
[Placeholder: A, B
          for i.0 in range(32):
                for j.0 in range(64):
          for ic.2 in range(16):
                for jc.2 in range(4):
                    for k.0 in range(2):
              for k.1 in range(16):
                   for k.2 in range(2):
                   for i.3 in range(2):
                   for j.3 in range(2):
                        C = … ]
```

```
Annotation 2:
[Placeholder: A, B
          for i.0 in range(2):
               for j.0 in range(1024):
          for ic.2 in range(32):
                for jc.2 in range(2):
                     for k.0 in range(2):
                for k.1 in range(8):
                      for k.2 in range(4):
                      for i.3 in range(4):
                      for j.3 in range(4):
                           C = … ]
```

Figure 5.2 Two annotations of the code sketch.

regression, and classification tasks of nodes and graphs [136,137]. There has been a surge of interest in GNN approaches for representation learning of graphs. GNNs follow a recursive neighborhood aggregation scheme, where each node aggregates feature vectors from its neighbors. After several iterations of aggregations, each node is represented by a new feature vector. The new feature vectors capture the structural information within the neighborhood [10, 70, 137]. The feature representation of the whole graph (*aka.*, graph embedding) is obtained via graph pooling (*aka.*, graph reduction, readout [87]), which maps the set of nodes into a compact representation o capture a meaningful and unified embedding of an entire graph. Mean pooling is a typical and widely used pooling method.

Transformers [122] have achieved great success and become the de facto choice for many natural language processing (NLP) tasks. Further, inspired by NLP successes, many works [20, 37] unveiled the potential of multi-head attention (MHA), which plays a crucial role in transformers for computer vision tasks. The attention operation has three inputs: query, key, and value vectors. Each query vector can attend to all the key vectors and compute the attention scores with respect to these key vectors. The final output is the weighted sum of the value vectors, with the attention scores as the weights. A significant advantage of MHA is that it can learn the long-range dependencies and complicated relationships between the inputs. The MHA stacks the attention modules to achieve outstanding performance in many fields [138, 157]. The readers may refer to [122] for more details on attention mechanisms and transformers.

5.3 Algorithms

5.3.1 Overall Flow of GTuner

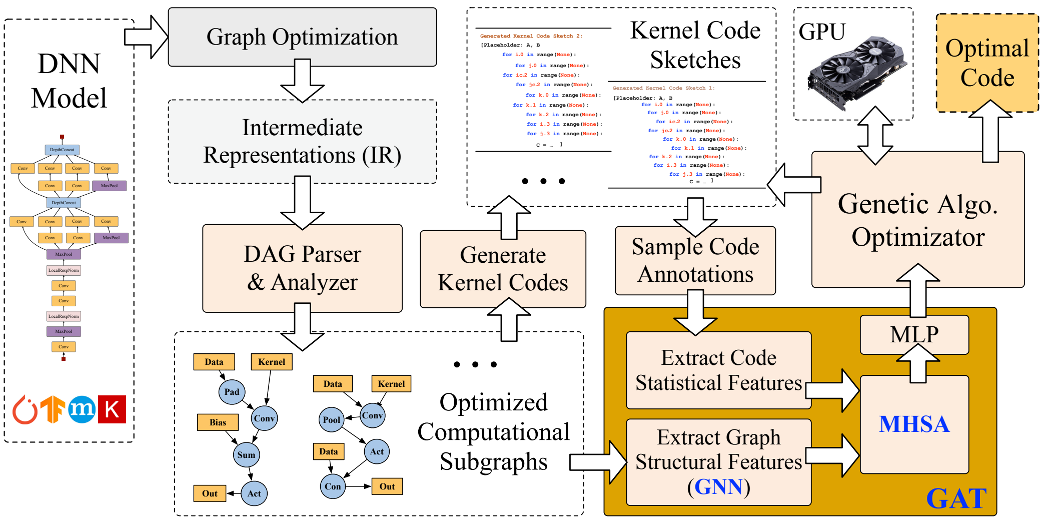


Figure 5.3 The overall flow of our GTuner.

The flow graph is shown in Fig. 5.3. Graph-level optimizations are conducted on the DNN model, *e.g.*, operator fusion, constant-folding, memory planning, data layout transformation, combinations of dense convolutions and linear operations, and operation canonicalizations. Then the optimized model is split into some subgraphs, *i.e.*, subtasks. GPU codes are optimized for each of these subgraphs independently. The final model deployment strategy is achieved by deploying these subgraphs sequentially [26].

After the graph optimizations, subgraphs are represented as intermediate representations (IRs) in the compilation tool. We implement a directed acyclic graph (DAG) parser and analyzer (lexical analysis and syntactic analysis) to analyze the IRs and construct the DAG representations for the subgraphs. Nodes in the DAG are computation and data nodes. Structural features are extracted for these nodes, including node types, shapes, operation types, *etc.* The node structural features are passed to a graph neural network to learn a unified embedding for the graph (*i.e.*, graph structural features). This embedding reflects this graph's scheduling and topological information, distinguishes different graphs, and improves the generalization to new tasks with various graphs structures.

Some rule-based kernel code templates (*i.e.*, sketches) are generated for each computational subgraph. The code sketches with parameters (*i.e.*, annotations) comprise the code design space of this subgraph. The statistical features are extracted via

Ansor and concatenated with the graph structural features for each sampled code. The concatenated features are then passed to the multi-head self-attention to learn the complicated relationships to predict the inference latency.

The genetic algorithm (GA) is adopted as the optimization method to explore the code design space to find the optimal code. Our GAT model predicts the inference latencies for the codes and guides the exploration of the GA algorithm. In each optimization iteration of the GA algorithm, codes sampled by GA are passed to GAT to predict the performance. Then the codes with the best-predicted performance are compiled and executed on GPU to get the actual performance. The final optimal code is selected according to the actual performance. The number of codes executed on GPU is also termed *measure trials*. For example, 10000 code annotations are sampled by GA and estimated by GAT. GAT selects the best 80 annotations and compiles and deploys them on GPU. Therefore, the measure trial is 80. The final deployment solution is the code with the best actual performance from these 80 codes.

In our framework, the graph level optimization, code generation, and genetic algorithm-based optimization are from TVM [26] and Ansor [153], and the other modules are ours.

Graph Attention Network (GAT) consists of the graph neural network (GNN) module and the multi-head self-attention (MHSA) module, as shown in Fig. 5.4. Structural analysis is conducted on the graph to get the structural features, including

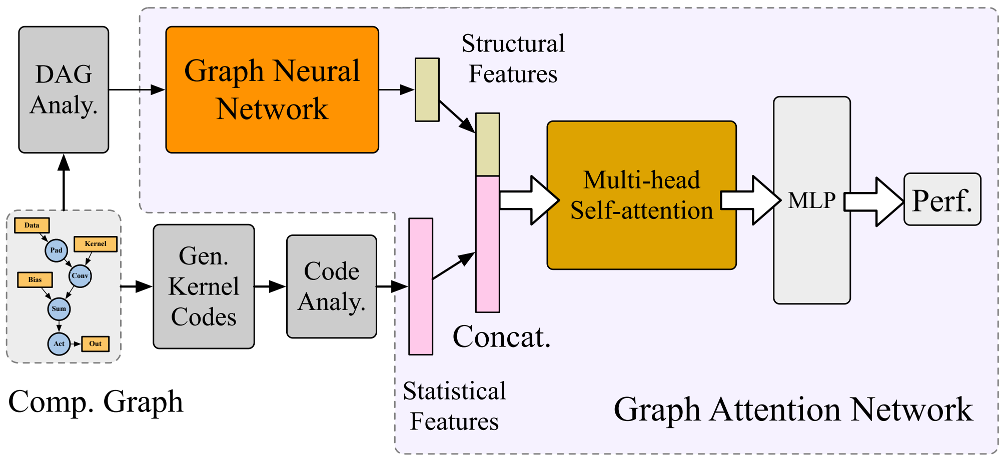


Figure 5.4 Structure of our graph attention network (GAT).

the types of operations, dimensions of data, *etc.* These features are the inputs to the graph neural network layers which will be discussed in Section 5.3.2. The output of the graph neural network is the structural features for the subgraph. The multi-head self-attention module will be discussed in detail in Section 5.3.3. Then a multi-layer perceptron (MLP) is appended to reduce the dimension and map to the performance.

5.3.2 GAT: Graph Neural Network Module

DNN models usually have different model structures, *e.g.*, the three structures shown in Fig. 5.5. The distinctive model structures result in different on-chip scheduling, communication, and computation patterns. These differences also help distinguish deployment tasks while their statistical features have no structural information. In other words, it is easy to identify the deployment tasks given structural information while the statistical information is incapable. In this section, the graph neural network is used to learn the representations for the subgraphs. Therefore, this part is independent of the sketches and annotations.

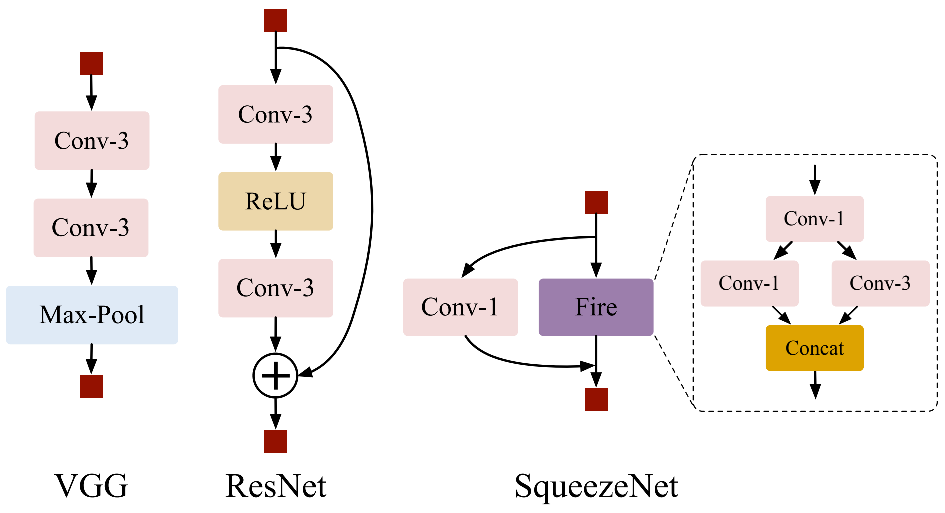


Figure 5.5 Typical diverse structures in VGG, ResNet (residual block) and SqueezeNet (fire block). Conv-x denotes that the kernel size is x.

Each computational subgraph to be implemented as a kernel is represented as $\mathcal{G}(\mathcal{V}, \mathcal{E})$, with the node set $\mathcal{V}$ and edge set $\mathcal{E}$. $\mathcal{V}$ has n nodes, *i.e.*, $|\mathcal{V}| = n$. The neighborhood set is $\mathcal{N}$, with $\mathcal{N}(v)$ denoting the neighbors of v. The features for the graph are represented as $\boldsymbol{X}$. Each vector $\boldsymbol{x}_i \in \boldsymbol{X}$ with $i \in \{1, \dots, n\}$ denotes the features for node $v_i \in \mathcal{V}$. Denote the feature length as d_x, *i.e.*, $\boldsymbol{x}_i \in \mathbb{R}^{1\times d_x}$. To learn the structural information of the computational subgraph, we update the representations of a node by aggregating information from its neighbors. Let AGGREGATE$(\cdot)$ denotes the aggregation function, and a function COMBINE combines the information from neighbors and the features of the node itself. These functions take the following common forms:

$$\begin{aligned}\boldsymbol{a}_i^k &= \text{AGGREGATE}^{(k)}(\{\boldsymbol{x}_t^{k-1} : v_t \in \mathcal{N}(v_i)\}),\\ \boldsymbol{x}_i^k &= \text{COMBINE}^{(k)}(\boldsymbol{x}_i^{k-1}, \boldsymbol{a}_i^k),\end{aligned} \tag{5.1}$$

where k represents the k-th iteration of aggregations in the graph neural network. For $k = 0$, $\boldsymbol{x}_i^{k=0}$ is the raw feature of node v_i itself.

Researchers have proposed many aggregation and combination functions for various applications [10,70,137]. An important target in our problem is to distinguish the different model structures. Therefore, the concept of the Weisfeiler-Leman (WL) test is of vital importance. The WL test is to distinguish the isomorphic graphs via information propagation and can identify the structural similarities between graphs. Further, [88] demonstrated that GNNs could be viewed as an extension of the WL test, which in principle have the same power but are more flexible in their ability to adapt to the learning task at hand and are able to handle complicated node features. We choose to use the graph convolutional layer used by [88], denoted as WL-GCN, in which the AGGREGATE and COMBINE steps are integrated. The function of the graph convolutional layer takes the form as follows:

$$\boldsymbol{x}_i^k = \boldsymbol{W}_1^{k-1}\boldsymbol{x}_i^{k-1} + \boldsymbol{W}_2^{k-1} \sum_{v_t \in \mathcal{N}(v_i)} \boldsymbol{x}_t^{k-1}, \tag{5.2}$$

where $\boldsymbol{W}_1^{k-1}$ and $\boldsymbol{W}_2^{k-1}$ denote the learnable weights. Then, the mean pooling is used to achieve the feature embedding for the

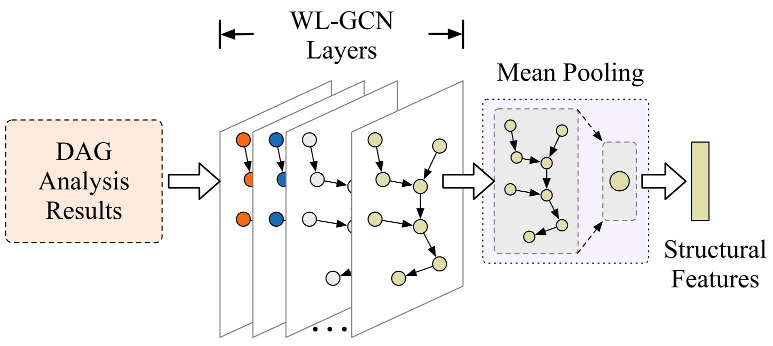


Figure 5.6 Graph neural network.

computational subgraph, *i.e.*, the feature vectors of the nodes in the graph are summed up and averaged as the feature for the graph:

$$\boldsymbol{G} = \frac{1}{n} \sum_{v_i \in \mathcal{V}} \boldsymbol{x}_i^K, \tag{5.3}$$

where $\boldsymbol{G}$ denotes the feature embedding for the subgraph, n is the number of nodes, and K is the maximum iteration of information aggregation. The model structure is shown in Fig. 5.6.

5.3.3 GAT: Multi-head Self-attention Module

The structural features learned by the graph neural network and the statistical features of the annotations should be analyzed to understand the complicated relationships between features, enhance the critical parts and fade out the unimportant parts. Unlike FPGA and TPU, there are no low-level implementation details on GPU. Therefore, determining which features are essential for the tasks should be finished automatically. As discussed above, the statistical features for GPU are weakly related to the computation patterns. In Ansor, the statistical features can be briefly categorized into some groups, *i.e.*, buffer access features, buffer storage features, arithmetic-related features, *etc.* Typical features include the number of touched cache lines, touched memory in bytes, number of floating multiplication operations, number of unrolled iterators, *etc.* They are unstructured data that are directly concatenated, with no information related to the kernel structures. The orders of the features in the vectors have no physical meanings. To build better performance models, it is necessary to learn the implicit relationships between the features while these are the prior knowledge for other types of devices. For simplicity of notations, the statistical and structural features after concatenation are briefly denoted as $\boldsymbol{x}$.

Inspired by self-attention mechanisms' success in the computer vision field, we propose to use the multi-head self-attention (MHSA) module to learn the features. To formulate MHSA, we first introduce the scaled dot-product attention $\mathrm{Attn}(\cdot)$. Given queries $\boldsymbol{Q} \in \mathbb{R}^{n_q \times d_k}$, keys $\boldsymbol{K} \in \mathbb{R}^{n \times d_k}$ and values $\boldsymbol{V} \in \mathbb{R}^{n \times d_v}$, we have [122]:

$$\mathrm{Attn}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right)\boldsymbol{V}, \tag{5.4}$$

where n_q, n, d_k, d_v are the query number, key number, key dimension and value dimension, respectively. Then we can define the multi-head attention as:

$$\mathrm{MHA}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = Concat(\boldsymbol{H}_1, \boldsymbol{H}_2, \cdots, \boldsymbol{H}_h)\boldsymbol{W}^O, \tag{5.5}$$

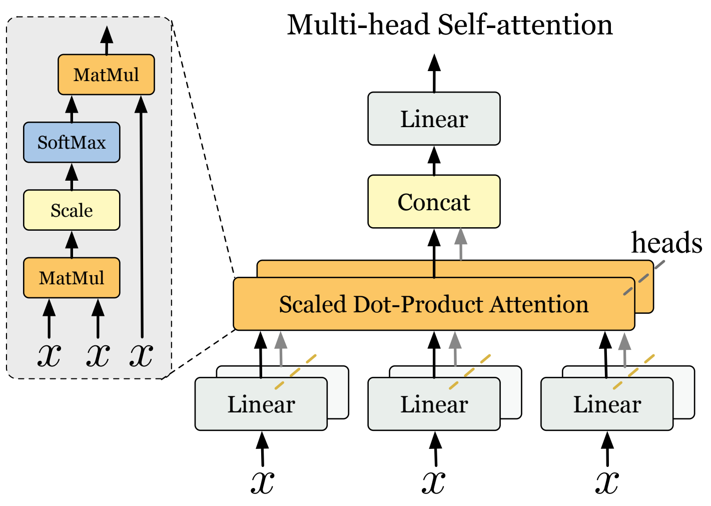


Figure 5.7 Multi-head self-attention with $\boldsymbol{x}$ as the query ($\boldsymbol{Q}$), key ($\boldsymbol{K}$), and value ($\boldsymbol{V}$) simultaneously.

where $\boldsymbol{H}_i$ is the output of the i-th attention head, h is the number of heads and $\boldsymbol{W}^O$ is the learnable projection weight matrix. $\boldsymbol{H}_i$ is computed by:

$$\boldsymbol{H}_i = \text{Attn}\left(\boldsymbol{Q}\boldsymbol{W}_i^Q, \boldsymbol{K}\boldsymbol{W}_i^K, \boldsymbol{V}\boldsymbol{W}_i^V\right), \tag{5.6}$$

where $\boldsymbol{W}_i^Q \in \mathbb{R}^{d_k \times d_k}$, $\boldsymbol{W}_i^K \in \mathbb{R}^{d_k \times d_k}$, $\boldsymbol{W}_i^V \in \mathbb{R}^{d_v \times d_v}$ are learnable projection matrices corresponding to the i-th head. Each query vector of $\boldsymbol{Q}$ can attend to all the key vectors of $\boldsymbol{K}$ and compute the attention scores concerning these key vectors. The final output is the weighted sum of the value vectors of $\boldsymbol{V}$, with attention scores as the weights. The output projection matrix is $\boldsymbol{W}^O \in \mathbb{R}^{hd_v \times d_{out}}$, where d_{out} is the output dimensionality for the multi-head attention function.

To fit the inputs of MHA, we firstly reshape the original vector $\boldsymbol{x}$ into several shorter vectors. Denote the original length of

$\boldsymbol{x}$ as l. Then the reshaped $\boldsymbol{x}$, denoted as $\boldsymbol{x}^R$, has the shape $h \times \frac{l}{h}$, where h is the number of heads in MHA. The $\boldsymbol{x}^R$ is used as the query ($\boldsymbol{Q}$), key ($\boldsymbol{K}$), and value ($\boldsymbol{V}$) simultaneously. MHA becomes **multi-head self-attention (MHSA)**, which captures global dependencies among the inputs, as shown in Fig. 5.7 and Equation (5.7).

$$\mathrm{SelfAttn}\left(\boldsymbol{x}^R\boldsymbol{W}_i^Q, \boldsymbol{x}^R\boldsymbol{W}_i^K, \boldsymbol{x}^R\boldsymbol{W}_i^V\right). \tag{5.7}$$

According to Equation (5.4), the inner products between vectors in $\boldsymbol{x}$ are computed. These products characterize the similarities between the feature vectors. Further, these similarities reflect the implicit relationships among the features, including the scheduling of the subgraphs, memory patterns, which features are related to each other or have similar influences on the performance, *etc.* The similarity values are then used as the weight scores to sum the values, *i.e.*, vectors in $\boldsymbol{x}^R$ itself. The connections between MHSA and inference latency are learned through model training. And the weights in MHSA will be updated to enhance the critical parts of the features and fade out the unimportant parts.

5.4 Experiments

In this section, we conduct experiments on our GTuner to validate the performance. The experimental inference platform is Nvidia GeForce RTX 3090 (Ampere architecture, SM86), with

CUDA Driver 11.4, PyTorch 1.10, and TVM 0.8-dev. The CPU is 16 core Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz. Some public models defined in TVM Relay library are tested, including ResNet-18, ResNet-34 [54], MobileNet V1 [58], and SqueezeNet V1.1 [59]. The representative DNN layers widely used in both industries and academia are all covered in these models, including conventional convolutional layers, shortcut layers, multi-branch layers, fully connected layers, depth-wise convolutional layers, *etc.* The tuning history of Ansor (with XGBoost as performance model) on Inception-V3 and VGG-11 is collected as the training data to train the models, in total about 170000 annotations. Some baselines are compared in the experiments to prove the effectiveness of our method, including Ansor [153], AutoTVM [28], PyTorch, and PyTorch with JIT optimization [92]. The performance metric is the end-to-end inference latency of the model. We also show the results of Giga floating operations per second (GFLOPS), which reflects the peak computation speed of a task on the device.

5.4.1 Implementation Details

Our GAT model has two WL-GCN layers [88] to process the graph features, a mean pooling layer, a concatenation layer (to concatenate statistical and graph features), a fully-connected layer (downscale the features to 512), a four-head multi-head self-attention layer (*i.e.*, $h = 4$), and an MLP regression module (with output dimensions: 200-100-20-1). In the benchmarks and Ansor, there are 136 elements representing the types of nodes, inputs, outputs, data shapes, types of operations, *etc.* The graph embedding has 136 elements after graph pooling. The statistical features reported by Ansor have 656 elements and are concatenated with the graph features and downscaled to 512 via the fully-connected layer.

The baseline multi-head self-attention (MHSA) comprises a four-head multi-head self-attention layer (the same as ours) and an MLP regression module (also the same as ours) without graph module and structural features.

We train the models with Adam optimizer for 300 epochs with a learning rate 1e-4 and batch size 512. The loss function is mean square error (MSE). The regression target is inference latency. We denote our method as GTuner (GNN + MHSA) in the result tables for ease of explanation.

Our method's genetic algorithm optimization process and the baselines all follow the default settings of TVM. The different numbers of measure trials are compared in the ablation studies. Note that except for the ablation studies on the measure trials, the measure trials of other results are 80 trials per subgraph. In other words, for a DNN model, the total number of measure trials is the product of the number of subgraphs and 80 trials.

5.4.2 Ablation Studies on GAT Structure

We compare our method with some widely-used graph convolutional layers, including spectral graph convolution (SpecGCN) [65], masked attention convolution (MaskGAT) [123], and GraphSAGE [50]. SpecGCN propagates information in the graph via a first-order approximation of localized spectral filters on the graphs. Learned filters are used to represent the nodes in the Fourier domain. MaskGAT introduces the attention-based architecture to compute the hidden representations of the nodes by using masks during information aggregation. GraphSAGE learns a function that generates embeddings by sampling and aggregating features from a node's local neighborhood to improve the generalization abilities to unseen nodes.

We replace the WL-GCN layers in GAT with these three convolutional layers. Under the same training and experimental settings, the inference latencies of ResNet-18 are listed in TABLE 5.1. The results show that these methods have poor performance in this problem. As mentioned before, graph isomorphism is important in our context. These methods fail to learn this kind of isomorphism and fall into the trap of overfitting to the training set and therefore are incapable of tackling new tasks and the isomorphic graphs with different volumes of computations and communications. SpecGCN outperforms Ansor ($1.016 < 1.073$, *i.e.*, 5.31%). MaskGAT and GraphSAGE force the model to ignore some information to improve the gen-

Table 5.1 Comparisons between Convolutional Layers

ResNet-18	Ansor [153]	GTuner	SpecGCN [65]	MaskGAT [123]	GraphSAGE [50]
Latency (ms)	1.073	0.923	1.016	1.105	1.168

Table 5.2 Performance without GNN or MHSA

ResNet-18	GTuner (GNN + MHSA)	MHSA	GNN + MLP
Latency (ms)	0.923	0.963	1.121

eralization. It degrades the performance significantly since the important topological and scheduling knowledge is discarded. These techniques achieve good results on big data problems, *e.g.*, social networks and recommendation systems with millions of nodes and edges. In our problem, the structures of graphs are limited compared with these applications, and it is unacceptable to forget information.

Further, to validate the performance of our proposed MHSA module, we do the ablation study via dropping the MHSA module in GAT. The outputs of the GNN module and the statistical features are concatenated and then directly passed to a fully-connected layer to achieve a feature vector with length 512. Then this feature vector is passed to the same MLP regression module. This experiment is denoted as GNN + MLP in TABLE 5.2. For comparison, the baseline MHSA is also listed. Results show that the combination of our GNN and MHSA achieves the best performance.

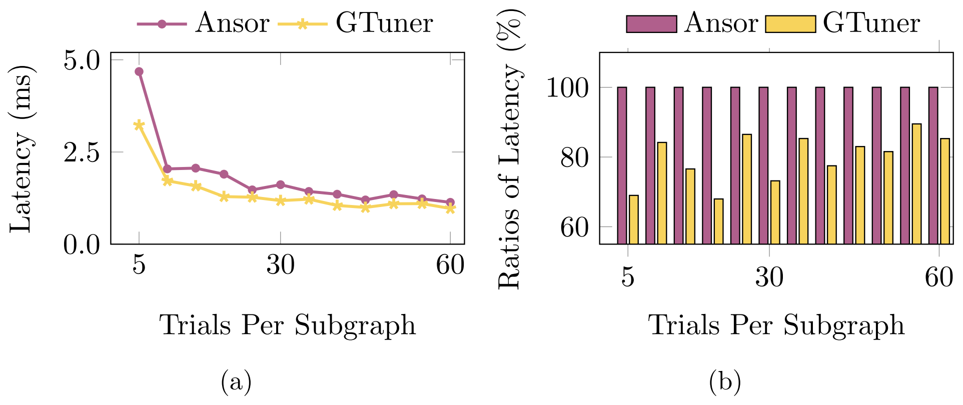
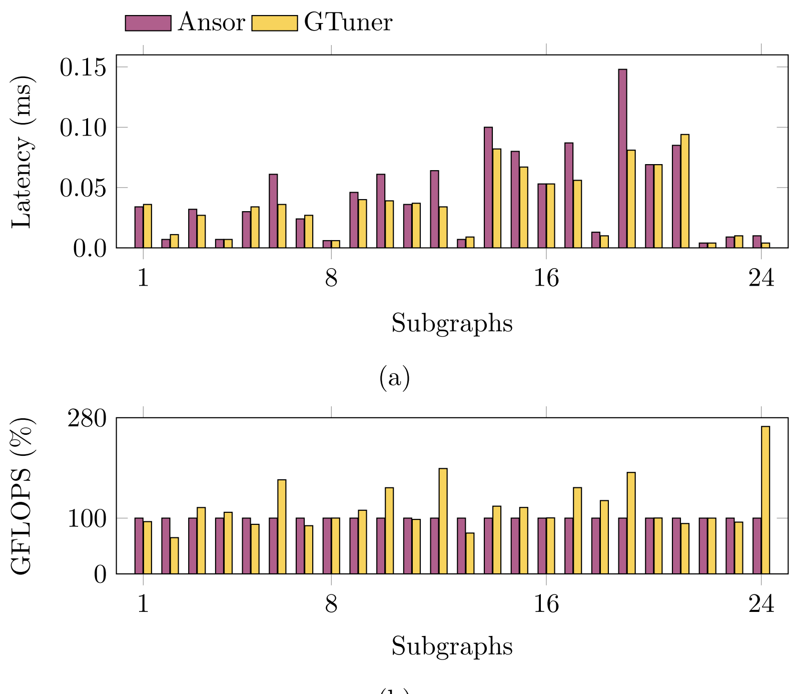


Figure 5.8 Results of different measure trials.

5.4.3 Ablation Studies on GPU Measure Trials

The genetic algorithm implemented by TVM is adopted as the default searching algorithm in Ansor and our method. Interacting with GPU helps validate the designs found by the searching algorithms. Therefore, more interactions (measure trials) will find better solutions. Some experiments are conducted on ResNet-18 to reveal the performance of our method under different measure trials, as shown in Fig. 5.8. The numbers of measure trials are 5, 10, 15, . . . , 60. The ratios of latencies with respect to Ansor are shown in Fig. 5.8(b). With the increases in measure trials, both Ansor and our GTuner can find better results, while our performance advantages compared with Ansor are still remarkable (more than 10%).



(b)

Figure 5.9 Detail results of subgraphs in ResNet-18.

5.4.4 Performance of the Whole framework

Detailed results of subgraphs in ResNet-18 are shown in Fig. 5.9, including the inference latencies and GFLOPS. The GFLOPS values are represented as ratios to Ansor. It is shown that our method reduces the latencies and improves the GFLOPS on most subgraphs. The performance improvements on the difficult subgraphs (with long latencies) are inspiring.

The end-to-end inference latencies of the DNN models are listed in TABLE 5.3. The results demonstrate the significant advantages of our method compared with the baselines. The proposed multi-head self-attention module improves the per-

Table 5.3 End-to-end Model Inference Latency (ms)

Model	PyTorch	PyTorch-JIT [92]	AutoTVM [28]	Ansor [153]	MHSA	**GTuner**
ResNet-18	27.180	4.119	1.056	1.073	0.963	**0.923** (**13.98%**)
ResNet-34	48.988	5.929	1.180	0.968	0.907	**0.872** (**9.92%**)
SqueezeNet	16.658	3.648	0.311	0.207	0.201	**0.197** (**4.83%**)
MobileNet	30.324	6.972	0.513	0.242	0.252	**0.227** (**6.20%**)

$^{+}$ Ratios are performance improvements compared with Ansor.

Table 5.4 Time Costs (minutes) of the Optimization Processes

Model	AutoTVM [28]	Ansor [153]	MHSA	GTuner
ResNet-18	45.57	45.95	46.94	65.22
ResNet-34	46.66	48.89	50.71	54.86
SqueezeNet	43.53	44.40	45.91	63.90
MobileNet	42.88	43.80	44.20	61.60

formance based on Ansor. The graph network module in our GTuner further improves the performance. On the four models in TABLE 5.3, our method reduces the latencies by **13.98**%, **9.92**%, **4.83**%, and **6.20**%, respectively, compared with Ansor, and more than **30%** on average compared with AutoTVM. The two PyTorch-based methods are rule-based, and their performance is not satisfying. The results also reveal that using the graph structural features helps improve the generalization of the performance model to new tasks that do not exist in the training set. The time costs of the whole optimization processes of TVM-based methods are listed in TABLE 5.4, including the searching processes of the genetic algorithm, the compilations of codes, and the interactions with GPU. It is shown that our performance improvements have no extra overheads on the time costs.

5.5 Conclusion

In this chapter, we proposes a novel method GTuner to optimize the computations of DNN models on GPU. The computational graphs are learned via the WL graph isomorphism convolutional layers to extract high-quality features. The structural and statistical features are jointly learned via the multi-head self-attention module to improve the regression quality. Our results outperform the baselines and prove the effectiveness of GTuner.

□ **End of chapter.**

Chapter 6

Conclusion

In this thesis, we have supplied some learning-based methodologies to stimulate the fast and efficient deployments of deep learning algorithms. In this chapter, we summarize the proposed methodologies and then discuss the future directions.

6.1 Summary

Deep learning deployments have been widely discussed in the AI era to facilitate the applications of deep learning algorithms. Though brilliant achievements have been made in the model design and training of neural networks, the gaps between these and the hardware-level implementations are still challenging. FPGA and GPU are the most popular hardware platforms for algorithm deployments due to their high parallelisms, high throughputs, and flexible configurability. In this thesis, three approaches are designed: one for FPGA-based and two for GPU-based implementations. Our methods are superior to prior deployment techniques on popular deep learning algorithms and models.

FPGA-based design flow is complicated and time-consuming, composed of three design stages, where each stage reports three performance metrics (power, latency, and resource consumption). High-level synthesis (HLS) is adopted to implement some code templates in Section 2.1 to ease the difficulties of FPGA implementations. Meanwhile, the HLS directives are left to be determined to improve the performance. The cumbersome design flow and the complex relationships between the HLS directives, the stages, and performance metrics make this problem nontrivial. In Chapter 3, we propose a correlated multi-objective multi-fidelity method to explore the Pareto-optimal HLS directives, which can balance the performance metrics well. Bayesian optimization is utilized as the optimization framework. An expected improvement of Pareto hypervolume is defined as the acquisition function, and the correlated Gaussian process models are adopted to characterize the complex relationships. The reports from the multiple stages are measured via the non-linear Gaussian process models. With these techniques, our method reduces optimization costs and simultaneously improves the quality of results.

Deployments on GPU suffer from the vast design space in which there are typically more than millions of candidate implementations for each layer, as discussed in Section 2.2. Though the compilations and executions of the GPU codes are usually shorter than several seconds, the optimizations are still expensive due to the extensive design spaces. The optimization processes start from scratch traditionally, while the historical optimization data are not utilized. In Chapter 4, we propose to use the deep Gaussian transfer learning methods to learn from the historical data. Maximum-a-posteriori (MAP) estimation is applied to tune the DGP model according to the history to make the DGP model accommodate the new task with no loss of hidden knowledge. With this method, we reduce the optimization costs remarkably and significantly improve the quality of results on the mainstreaming deep learning models on Nvidia GPUs.

Furthermore, researchers devise advanced code generation rules to realize larger design spaces covering better code implementations. Despite the implementation-level progress, model structural information is ignored in the performance estimator, degrading the performance accuracies in the exploration process. Besides, the mutual feature effects and importance are well modeled so as to result in poor performance. In Chapter 5, we use a graph neural network to delve into the neural network model structures and design a multi-head self-attention module to enhance the critical parts of the features and fade out the unimportant parts. The experiments on deep learning algorithms demonstrate inspiring improvements.

6.2 Possible Future Directions

Specific devices for deep learning computations have received growing attention in recent years, and the joint optimization of the models and the hardware platforms is a prevailing direction. In optimizations, the collaborative design for the models and hardware enlarges the design space and prompts us to use more sophisticated optimization methods. Differentiable model structure optimization is helpful to update the model structures together with the model parameters according to the training or testing loss and avoid ineffective structure explorations. To deal with the explosive memory costs, we must introduce more techniques while guaranteeing a fast and accurate performance estimation for the new structures. Besides, the hardware architectures should be designed in alignment with the model structures. Consequently, we can devise some architectural templates to form the hardware design spaces. In the joint optimization, some crucial questions must be solved, including the definitions of the design spaces, high-quality performance models, and efficient exploration methods.

Heterogeneous computing for the deep learning algorithms promises thanks to the growing support from the hardware vendors. Modern devices usually integrate various hardware resources to facilitate their usage in diverse scenarios. Therefore, the existing deep learning deployment techniques that rely on a single device suffer from low system utilization. Further, some researchers explore the potential of integrating various platforms to take full advantage of various technological advancements. [106] couples an LSTM co-processor with an embedded RISC-V CPU in the RISC-V design flow to compute sparse LSTM tasks which are for character recognition and language models. [49] builds a simultaneous multi-mode architecture that integrates the systolic model with a GPU-like SIMD execution model and outperforms the general processors CPU and GPU significantly. For graph neural networks, [112] designs a framework that contains a dense engine to compute fully connected (FC) layers and a graph engine to compute the message passing in the graph. Heterogeneous computing can tackle more complicated deep learning models such as the super-resolution (SR) models. Unlike the widely-discussed deep learning models, the SR algorithms are composed of irregular operations and complicated structures that require specific optimizations. Compared with the well-studied models, such as VGG, in the SR models, shift operations, shuffles, tensor multiplications, long-term dependencies, *etc.*, make the optimization task non-trivial. These operations are not considered in the general deployment frameworks for classification or regression deep learning models. The SR model is divided into some small segments, thus making the existing tools fail to achieve the end-to-end inference. Besides, the volumes of frames increase in the model, resulting in heavier communication workloads. With the heterogeneous platforms, the irregular tasks can be computed by the specifically designed

FPGA kernels or the onboard CPUs, and the common tasks are deployed on AI engines or GPUs. Considering these circumstances, we need to explore the task analysis, allocation, system integration, fast deployment flows, and provide easy-to-use computation libraries, optimization techniques, and interfaces to ease designers from cumbersome developments.

□ **End of chapter.**

Bibliography

[1] NVIDIA-TensorRT. https://developer.nvidia.com/zh-cn/tensorrt.

[2] oneAPI Deep Neural Network Library (oneDNN). https://www.oneapi.io/open-source/.

[3] Pragma HLS Array Partition. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-array_partition.

[4] Vivado Design Suite User Guide: High-Level Synthesis. https://www.xilinx.com/support/documents/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf.

[5] Xilinx Deep Processing Unit (DPU). https://www.xilinx.com/html_docs/xilinx2019_2/vitis_doc/dpu_over.html.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, et al. TensorFlow: A system for large-scale machine learning. In *Proc. OSDI*, pages 265–283, 2016.

[7] B. H. Ahn, P. Pilligundla, A. Yazdanbakhsh, and H. Esmaeilzadeh. CHAMELEON: Adaptive code optimization for expedited deep neural network compilation. In *Proc. ICLR*, 2020.

[8] T. Ajanthan, P. K. Dokania, R. Hartley, and P. H. Torr. Proximal mean-field for neural network quantization. In *Proc. ICCV*, pages 4871–4880, 2019.

[9] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*, 2016.

[10] J. Atwood and D. Towsley. Diffusion-convolutional neural networks. In *Proc. NeurIPS*, pages 1993–2001, 2016.

[11] M. Balandat, B. Karrer, D. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy. BoTorch: A framework for efficient Monte-Carlo Bayesian optimization. In *Proc. NeurIPS*, volume 33, pages 21524–21538, 2020.

[12] S. Bansal, H. Hsiao, T. Czajkowski, and J. H. Anderson. High-level synthesis of software-customizable floating-point cores. In *Proc. DATE*, pages 37–42, 2018.

[13] S. Belakaria, A. Deshwal, and J. R. Doppa. Max-value entropy search for multi-objective Bayesian optimization. In *Proc. NeurIPS*, volume 32, pages 7825–7835, 2019.

[14] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

[15] E. V. Bonilla, K. M. Chai, and C. Williams. Multi-task gaussian process prediction. In *Proc. NeurIPS*, pages 153–160, 2008.

[16] J. Brownlee. *Data preparation for machine learning: data cleaning, feature selection, and data transforms in Python.* Machine Learning Mastery, 2020.

[17] T. Bui, D. Hernández-Lobato, J. Hernandez-Lobato, Y. Li, and R. Turner. Deep Gaussian processes for regression using approximate expectation propagation. In *Proc. ICML*, pages 1472–1481, 2016.

[18] H. Cai et al. ProxylessNAS: Direct neural architecture search on target task and hardware. In *Proc. ICLR*, 2019.

[19] R. Calandra, J. Peters, C. E. Rasmussen, and M. P. Deisenroth. Manifold Gaussian processes for regression. In *Proc. IJCNN*, pages 3338–3345. IEEE, 2016.

[20] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko. End-to-end object detection with transformers. In *Proc. ECCV*, 2020.

[21] J.-W. Chang, S. Ahn, K.-W. Kang, and S.-J. Kang. Towards design methodology of efficient fast algorithms for

accelerating generative adversarial networks on FPGAs. In *Proc. ASPDAC*, pages 283–288. IEEE, 2020.

[22] T. Chen, B. Duan, Q. Sun, M. Zhang, G. Li, H. Geng, Q. Zhang, and B. Yu. An efficient sharing grouped convolution via Bayesian learning. *IEEE TNNLS*, pages 1–13, 2021.

[23] T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *Proc. KDD*, pages 785–794, 2016.

[24] T. Chen, B. Lin, H. Geng, S. Hu, and B. Yu. Leveraging spatial correlation for sensor drift calibration in smart building. *IEEE TCAD*, 40(7):1273–1286, 2021.

[25] T. Chen, B. Lin, H. Geng, and B. Yu. Sensor drift calibration via spatial correlation model in smart building. In *Proc. DAC*, pages 1–6, 2019.

[26] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proc. OSDI*, pages 578–594, 2018.

[27] T. Chen, Q. Sun, and B. Yu. Machine learning in nanometer AMS design-for-reliability (invited paper). In *Proc. ASICON*, pages 1–4, 2021.

[28] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy. Learning to opti-

mize tensor programs. In *Proc. NeurIPS*, pages 3389–3400, 2018.

[29] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *Proc. ISSCC*, pages 262–263, 2016.

[30] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE JETCAS*, 9(2):292–308, 2019.

[31] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint*, 2014.

[32] K. Choi, D. Hong, H. Yoon, J. Yu, Y. Kim, and J. Lee. DANCE: Differentiable accelerator/network co-exploration. In *Proc. DAC*, pages 337–342. IEEE, 2021.

[33] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Young, and Z. Zhang. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *Proc. FCCM*, pages 129–132, 2018.

[34] A. G. de G. Matthews, J. Hron, M. Rowland, R. E. Turner, and Z. Ghahramani. Gaussian process behaviour in wide deep neural networks. In *Proc. ICLR*, 2018.

[35] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proc. NAACL*, pages 4171–4186, 2019.

[36] Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer. HAWQ: Hessian aware quantization of neural networks with mixed-precision. In *Proc. ICCV*, pages 293–302, 2019.

[37] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proc. ICLR*, 2021.

[38] Ł. Dudziak, T. Chau, M. S. Abdelfattah, R. Lee, H. Kim, and N. D. Lane. BRP-NAS: Prediction-based NAS using GCNs. In *Proc. NeurIPS*, 2020.

[39] D. Duvenaud, J. Lloyd, R. Grosse, J. Tenenbaum, and G. Zoubin. Structure discovery in nonparametric regression through compositional kernel search. In *Proc. ICML*, pages 1166–1174, 2013.

[40] H. Fan, M. Ferianc, M. Rodrigues, H. Zhou, X. Niu, and W. Luk. High-performance FPGA-based accelerator for bayesian neural networks. In *Proc. DAC*, 2021.

[41] J. Fang, A. Shafiee, H. Abdel-Aziz, D. Thorsley, G. Georgiadis, and J. H. Hassoun. Post-training piecewise linear quantization for deep neural networks. In *Proc. ECCV*, pages 69–86, 2020.

[42] L. Ferretti, G. Ansaloni, and L. Pozzi. Lattice-traversing design space exploration for high level synthesis. In *Proc. ICCD*, pages 210–217, 2018.

[43] J. Gardner, G. Pleiss, K. Q. Weinberger, D. Bindel, and A. G. Wilson. GPyTorch: Blackbox matrix-matrix Gaussian process inference with GPU acceleration. In *Proc. NeurIPS*, pages 7587–7597, 2018.

[44] H. Geng, Y. Ma, Q. Xu, J. Miao, S. Roy, and B. Yu. High-speed adder design space exploration via graph neural processes. *IEEE TCAD*, pages 1–1, 2021.

[45] H. Geng, H. Yang, Y. Ma, J. Mitra, and B. Yu. SRAF insertion via supervised dictionary learning. In *Proc. ASPDAC*, pages 406–411, 2019.

[46] A. Ghoshal and J. Honorio. Learning maximum-a-posteriori perturbation models for structured prediction in polynomial time. In *Proc. ICML*, pages 1754–1762. PMLR, 2018.

[47] C. Gong, Z. Jiang, D. Wang, Y. Lin, Q. Liu, and D. Z. Pan. Mixed precision neural architecture search for energy

efficient deep learning. In *Proc. ICCAD*, pages 1–7. IEEE, 2019.

[48] I. Goodfelow, Y. Bengio, and A. Courville. Deep learning (adaptive computation and machine learning series), 2016.

[49] C. Guo, Y. Zhou, J. Leng, Y. Zhu, Z. Du, Q. Chen, C. Li, B. Yao, and M. Guo. Balancing efficiency and flexibility for DNN acceleration via temporal GPU-systolic array integration. In *Proc. DAC*, pages 1–6. IEEE, 2020.

[50] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Proc. NeurIPS*, pages 1024–1034, 2017.

[51] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *Proc. ICLR*, 2016.

[52] 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 *Proc. DAC*, pages 1–6, 2019.

[53] T. Hazan, F. Orabona, A. D. Sarwate, S. Maji, and T. S. Jaakkola. High dimensional inference with random maximum a-posteriori perturbations. *IEEE Transactions on Information Theory (TIT)*, 65(10):6539–6560, 2019.

[54] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proc. CVPR*, pages 770–778, 2016.

[55] Z. He, H. Huang, M. Jiang, Y. Bai, and G. Luo. FPGA-based real-time super-resolution system for ultra high definition videos. In *Proc. FCCM*, pages 181–188. IEEE, 2018.

[56] D. Hernández-Lobato, J. Hernandez-Lobato, A. Shah, and R. Adams. Predictive entropy search for multi-objective Bayesian optimization. In *Proc. ICML*, pages 1492–1501. PMLR, 2016.

[57] J. M. Hernández-Lobato, M. W. Hoffman, and Z. Ghahramani. Predictive entropy search for efficient global optimization of black-box functions. In *Proc. NeurIPS*, volume 27, pages 918–926, 2014.

[58] 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.

[59] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[60] iSmartDNN. `https://github.com/onioncc/iSmartDNN`.

[61] 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 *Proc. ISCA*, pages 1–12, 2017.

[62] M. Kandemir. Asymmetric transfer learning with deep gaussian processes. In *Proc. ICML*, pages 730–738, 2015.

[63] S. J. Kaufman, P. M. Phothilimthana, Y. Zhou, C. Mendis, S. Roy, A. Sabne, and M. Burrows. A learned performance model for tensor processing units. 2021.

[64] M. E. E. Khan, A. Immer, E. Abedi, and M. Korzepa. Approximate inference turns deep networks into gaussian processes. In *Proc. NeurIPS*, volume 32, pages 3094–3104, 2019.

[65] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *Proc. ICLR*, 2017.

[66] D. Kirk, B. S. Center, et al. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[67] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proc. NeurIPS*, pages 1097–1105, 2012.

[68] W. Kwon, G.-I. Yu, E. Jeong, and B.-G. Chun. Nimble: Lightweight and parallel GPU task scheduling for deep learning. In *Proc. NeurIPS*, 2020.

[69] J. Lee, Y. Bahri, R. Novak, S. S. Schoenholz, J. Pennington, and J. Sohl-Dickstein. Deep neural networks as Gaussian processes. In *Proc. ICLR*, 2018.

[70] J. Lee, I. Lee, and J. Kang. Self-attention graph pooling. In *Proc. ICML*, pages 3734–3743. PMLR, 2019.

[71] R. Li, Y. Xu, A. Sukumaran-Rajam, A. Rountev, and P. Sadayappan. Analytical characterization and design space exploration for optimization of CNNs. In *Proc. ASPLOS*, pages 928–942, 2021.

[72] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley. Differentiable programming for image processing and deep learning in Halide. In *ACM SIGGRAPH*, volume 37, pages 139:1–139:13, 2018.

[73] 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. In *Proc. DAC*, pages 1–6. IEEE, 2020.

[74] Z. Li, T.-Y. Ho, K. Y.-T. Lai, K. Chakrabarty, P.-H. Yu, and C.-Y. Lee. High-level synthesis for micro-electrode-

dot-array digital microfluidic biochips. In *Proc. DAC*, pages 1–6, 2016.

[75] Z. Lin, J. Zhao, S. Sinha, and W. Zhang. HL-Pow: A learning-based power modeling framework for high-level synthesis. In *Proc. ASPDAC*, pages 574–580. IEEE, 2020.

[76] H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable architecture search. In *Proc. ICLR*, 2018.

[77] H.-Y. Liu and L. P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *Proc. DAC*, pages 1–7, 2013.

[78] S. Liu, F. Lau, and B. C. Schafer. Accelerating FPGA prototyping through predictive model-based HLS design space exploration. In *Proc. DAC*, pages 1–6, 2019.

[79] S. Liu, Q. Sun, P. Liao, Y. Lin, and B. Yu. Global placement with deep learning-enabled explicit routability optimization. In *Proc. DATE*, pages 1821–1824, 2021.

[80] X. Liu, Y. Chen, C. Hao, A. Dhar, and D. Chen. WinoCNN: Kernel sharing Winograd systolic array for efficient convolutional neural network acceleration on FPGAs. In *Proc. ASAP*, pages 258–265. IEEE, 2021.

[81] Z. Liu, H. Mu, X. Zhang, Z. Guo, X. Yang, K.-T. Cheng, and J. Sun. Metapruning: Meta learning for automatic

neural network channel pruning. In *Proc. ICCV*, pages 3296–3305, 2019.

[82] C. Lo and P. Chow. Model-based optimization of high-level synthesis directives. In *Proc. FPL*, pages 1–10, 2016.

[83] C. Lo and P. Chow. Multi-fidelity optimization for high-level synthesis directives. In *Proc. FPL*, pages 272–277, 2018.

[84] C. Louizos, K. Ullrich, and M. Welling. Bayesian compression for deep learning. In *Proc. NeurIPS*, volume 30, 2017.

[85] W. Lyu, F. Yang, C. Yan, D. Zhou, and X. Zeng. Batch Bayesian optimization via multi-objective acquisition ensemble for automated analog circuit design. In *Proc. ICML*, volume 80, pages 3312–3320, 10–15 Jul 2018.

[86] A. Mahapatra and B. C. Schafer. Machine-learning based simulated annealer method for high level synthesis design space exploration. In *Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn)*, pages 1–6. IEEE, 2014.

[87] D. Mesquita, A. Souza, and S. Kaski. Rethinking pooling in graph neural networks. In *Proc. NeurIPS*, volume 33, pages 2220–2231, 2020.

[88] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. Weisfeiler and Leman go neural: Higher-order graph neural networks. In *Proc. AAAI*, volume 33, pages 4602–4609, 2019.

[89] J. Mu, M. Wang, L. Li, J. Yang, W. Lin, and W. Zhang. A history-based auto-tuning framework for fast and high-performance DNN design on GPU. In *Proc. DAC*, pages 1–6. IEEE, 2020.

[90] S. W. Ober, C. E. Rasmussen, and M. van der Wilk. The promises and pitfalls of deep kernel learning. In *Proc. UAI*, volume 161, pages 1206–1216. PMLR, 27–30 Jul 2021.

[91] K. O'Neal, M. Liu, H. Tang, A. Kalantar, K. DeRenard, and P. Brisk. HLSPredict: cross platform performance prediction for FPGA high-level synthesis. In *Proc. ICCAD*, pages 1–8, 2018.

[92] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Workshop*, 2017.

[93] M. Patacchiola, J. Turner, E. J. Crowley, M. O'Boyle, and A. Storkey. Deep kernel transfer in Gaussian processes for few-shot learning. *arXiv preprint arXiv:1910.05199*, 2019.

[94] A. Prost-Boucle, O. Muller, and F. Rousseau. A fast and autonomous HLS methodology for hardware accelerator generation under resource constraints. In *Euromicro Conference on Digital System Design*, pages 201–208. IEEE, 2013.

[95] S. Qu, B. Li, Y. Wang, D. Xu, X. Zhao, and L. Zhang. RaQu: An automatic high-utilization CNN quantization and mapping framework for general-purpose RRAM accelerator. In *Proc. DAC*, pages 1–6. IEEE, 2020.

[96] J. Quinonero-Candela and C. E. Rasmussen. A unifying view of sparse approximate Gaussian process regression. *The Journal of Machine Learning Research*, 6:1939–1959, 2005.

[97] C. E. Rasmussen and C. Williams. Gaussian processes for machine learning. *Cambridge, MA*, 32:68, 2006.

[98] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 110–119, 2014.

[99] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Proc. NeurIPS*, pages 91–99, 2015.

[100] H. Salimbeni and M. Deisenroth. Doubly stochastic variational inference for deep Gaussian processes. In *Proc. NeurIPS*, pages 4588–4599, 2017.

[101] H. Sapkota, Y. Ying, F. Chen, and Q. Yu. Distributionally robust optimization for deep kernel multiple instance learning. In *Proc. AISTATS*, volume 130, pages 2188–2196. PMLR, 2021.

[102] B. C. Schafer and K. Wakabayashi. Design space exploration acceleration through operation clustering. *IEEE TCAD*, 29(1):153–157, 2009.

[103] B. C. Schafer and K. Wakabayashi. Divide and conquer high-level synthesis design space exploration. *ACM TODAES*, 17(3):1–19, 2012.

[104] B. C. Schafer and Z. Wang. High-level synthesis design space exploration: past, present, and future. *IEEE TCAD*, 39(10):2628–2639, 2019.

[105] A. Shah and Z. Ghahramani. Pareto frontier learning with expensive correlated objectives. In *Proc. ICML*, volume 48, pages 1919–1927, 2016.

[106] R. Shi, J. Liu, K.-H. H. So, S. Wang, and Y. Liang. E-LSTM: Efficient inference of sparse LSTM on embedded heterogeneous system. In *Proc. DAC*, pages 1–6. IEEE, 2019.

[107] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In *Proc. CVPR*, pages 1874–1883, 2016.

[108] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proc. ICLR*, 2015.

[109] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Proc. NeurIPS*, volume 25, pages 2951–2959, 2012.

[110] H. F. Song, A. Abdolmaleki, J. T. Springenberg, A. Clark, H. Soyer, J. W. Rae, S. Noury, A. Ahuja, S. Liu, D. Tirumala, N. Heess, D. Belov, M. Riedmiller, and M. M. Botvinick. V-MPO: On-policy maximum a posteriori policy optimization for discrete and continuous control. In *Proc. ICLR*, 2020.

[111] Z. Song, J. Wang, T. Li, L. Jiang, J. Ke, X. Liang, and N. Jing. GPNPU: enabling efficient hardware-based direct convolution with multi-precision support in GPU tensor cores. In *Proc. DAC*, pages 1–6. IEEE, 2020.

[112] J. R. Stevens, D. Das, S. Avancha, B. Kaul, and A. Raghunathan. GNNerator: A hardware/software framework for accelerating graph neural networks. In *Proc. DAC*, 2021.

[113] Q. Sun, C. Bai, T. Chen, H. Geng, X. Zhang, Y. Bai, and B. Yu. Fast and efficient DNN deployment via deep Gaussian transfer learning. In *Proc. ICCV*, pages 5380–5390, October 2021.

[114] Q. Sun, C. Bai, H. Geng, and B. Yu. Deep neural network hardware deployment optimization via advanced active learning. In *Proc. DATE*, pages 1510–1515, 2021.

[115] Q. Sun, T. Chen, S. Liu, J. Chen, H. Yu, and B. Yu. Correlated multi-objective multi-fidelity optimization for hls directives design. *ACM TODAES*, 27(4), mar 2022.

[116] Q. Sun, T. Chen, J. Miao, and B. Yu. Power-driven DNN dataflow optimization on FPGA. In *Proc. ICCAD*, pages 1–7, 2019.

[117] Q. Sun, T. Chen, L. Siting, J. Miao, J. Chen, H. Yu, and B. Yu. Correlated multi-objective multi-fidelity optimization for HLS directives design. In *Proc. DATE*, pages 46–51, 2021.

[118] Q. Sun, X. Yao, A. A. Rao, B. Yu, and S. Hu. Counteracting adversarial attacks in autonomous driving. *IEEE TCAD*, 2022.

[119] Q. Sun, X. Zhang, H. Geng, Y. Zhao, Y. Bai, H. Zheng, and B. Yu. GTuner: Tuning DNN computations on GPU via graph attention network. In *Proc. DAC*, 2022.

[120] H. Tian, B. Liu, X.-T. Yuan, and Q. Liu. Meta-learning with network pruning. In *Proc. ECCV*, pages 675–700, 2020.

[121] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang. Lamda: Learning-assisted multi-stage autotuning for FPGA design closure. In *Proc. FCCM*, pages 74–77, 2019.

[122] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Proc. NeurIPS*, 2017.

[123] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *Proc. ICLR*, 2018.

[124] M. Volpp, L. P. Fröhlich, K. Fischer, A. Doerr, S. Falkner, F. Hutter, and C. Daniel. Meta-learning acquisition functions for transfer learning in Bayesian optimization. In *Proc. ICLR*, 2020.

[125] F. Wang, P. Cachecho, W. Zhang, S. Sun, X. Li, R. Kanj, and C. Gu. Bayesian model fusion: large-scale performance modeling of analog and mixed-signal circuits by reusing early-stage data. *IEEE TCAD*, 35(8):1255–1268, 2016.

[126] Z. Wang and S. Jegelka. Max-value entropy search for efficient Bayesian optimization. In *Proc. ICML*, volume 70, pages 3627–3635. PMLR, 2017.

[127] X. Wei, Y. Liang, and J. Cong. Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management. In *Proc. DAC*, pages 125–1, 2019.

[128] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proc. DAC*, pages 1–6, 2017.

[129] L. Wenbo, Z. Kun, Q. Lu, J. Nianjuan, L. Jiangbo, and J. Jiaya. LAPAR: Linearly-assembled pixel-adaptive regression network for single image super-resolution and beyond. In *Proc. NeurIPS*, 2020.

[130] L. While, P. Hingston, L. Barone, and S. Huband. A faster algorithm for calculating Hypervolume. *IEEE Transactions on Evolutionary Computation*, 10(1):29–38, 2006.

[131] A. Wilson and H. Nickisch. Kernel interpolation for scalable structured Gaussian processes (KISS-GP). In *Proc. ICML*, volume 37, pages 1775–1784. PMLR, 2015.

[132] A. G. Wilson, Z. Hu, R. Salakhutdinov, and E. P. Xing. Deep kernel learning. In *Proc. AISTATS*, pages 370–378, 2016.

[133] A. G. Wilson, Z. Hu, R. R. Salakhutdinov, and E. P. Xing. Stochastic variational deep kernel learning. In *Proc. NeurIPS*, volume 29, pages 2586–2594, 2016.

[134] J. T. Wilson, F. Hutter, and M. P. Deisenroth. Maximizing acquisition functions for Bayesian optimization. In *Proc. NeurIPS*, volume 31, pages 9884–9895, 2018.

[135] J. T. Wilson, R. Moriconi, F. Hutter, and M. P. Deisenroth. The reparameterization trick for acquisition functions. In *Workshop on Bayesian Optimization of Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[136] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip. A comprehensive survey on graph neural networks. *IEEE TNNLS*, 32(1):4–24, 2020.

[137] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *Proc. ICLR*, 2019.

[138] S. Yang, Z. Quan, M. Nie, and W. Yang. Transpose: Keypoint localization via transformer. In *Proc. ICCV*, pages 11802–11812, 2021.

[139] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen. HybridDNN: A framework for high-performance hybrid DNN accelerator design and implementation. In *Proc. DAC*, 2020.

[140] Z. Yin, W. Gross, and B. H. Meyer. Probabilistic sequential multi-objective optimization of convolutional neural networks. In *Proc. DATE*, pages 1055–1060. IEEE, 2020.

[141] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram. Virtual thread: Maximizing thread-level parallelism beyond GPU scheduling limit. In *Proc. ISCA*, pages 609–621, 2016.

[142] J. Yu, Y. Fan, J. Yang, N. Xu, Z. Wang, X. Wang, and T. Huang. Wide activation for efficient and accurate image super-resolution. In *Proc. BMVC*, 2019.

[143] Z. Yuan, B. Wu, G. Sun, Z. Liang, S. Zhao, and W. Bi. S2DNAS: Transforming static CNN model for dynamic inference via neural architecture search. In *Proc. ECCV*, pages 175–192, 2020.

[144] P. Zhang, E. Lo, and B. Lu. High performance depthwise and pointwise convolutions on mobile devices. In *Proc. AAAI*, pages 6795–6802, 2020.

[145] X. Zhang, Y. Li, C. Hao, K. Rupnow, J. Xiong, W.-m. Hwu, and D. Chen. SkyNet: A champion model for DAC-SDC on low power object detection. *arXiv preprint*, 2019.

[146] X. Zhang, Y. Ma, J. Xiong, W.-m. Hwu, V. Kindratenko, and D. Chen. Exploring HW/SW co-design for video analysis on CPU-FPGA heterogeneous systems. *IEEE TCAD*, 2021.

[147] 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 *Proc. ICCAD*, pages 56:1–56:8, 2018.

[148] X. Zhang, B. Zhu, X. Yao, Q. Sun, R. Li, and B. Yu. Context-based contrastive learning for scene text recognition. In *Proc. AAAI*, 2022.

[149] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *Proc. ICCAD*, pages 430–437, 2017.

[150] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He. Performance modeling and directives optimization for high level synthesis on FPGA. *IEEE TCAD*, 39(7):1428–1441, 2019.

[151] J. Zhao, T. Liang, S. Sinha, and W. Zhang. Machine learning based routing congestion prediction in FPGA high-level synthesis. In *Proc. DATE*, pages 1130–1135, 2019.

[152] W. Zhao, Q. Sun, Y. Bai, W. Li, H. Zheng, B. Yu, and M. D. Wong. A high-performance accelerator for super-resolution processing on embedded GPU. In *Proc. ICCAD*, 2021.

[153] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *Proc. OSDI*, pages 863–879, 2020.

[154] L. Zheng, R. Liu, J. Shao, T. Chen, J. E. Gonzalez, I. Stoica, and A. H. Ali. TenSet: A large-scale program performance dataset for learned tensor compilers. In *Proc. NeurIPS*, 2021.

[155] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar. Lin-analyzer: a high-level performance analysis tool for FPGA-based accelerators. In *Proc. DAC*, pages 1–6, 2016.

[156] Y. Zhou, Y. Zhang, Y. Wang, and Q. Tian. Accelerate CNN via recursive Bayesian pruning. In *Proc. ICCV*, pages 3306–3315, 2019.

[157] B. Zhu, R. Chen, X. Zhang, F. Yang, X. Zeng, B. Yu, and M. D. Wong. Hotspot detection via multi-task learning

and transformer encoder. In *Proc. ICCAD*, pages 1–8, 2021.

[158] W. Zuo, W. Kemmerer, J. B. Lim, L.-N. Pouchet, A. Ayupov, T. Kim, K. Han, and D. Chen. A polyhedral-based systemc modeling and generation framework for effective low-power design space exploration. In *Proc. ICCAD*, pages 357–364. IEEE, 2015.