ABSTRACT

Title of Dissertation: EFFICIENT SIMULATION AND IMPLEMENTATION OF NEURAL NETWORKS ON RESOURCE-CONSTRAINED PLATFORMS

Lei Pan, Doctor of Philosophy, 2023

Dissertation Directed by: Professor Shuvra S. Bhattacharyya
Dept. of Electrical and Computer Engr., and
Institute for Advanced Computer Studies

Neural Networks have been widely adopted in signal processing applications and systems. Due to the application scenarios enabled by the portability and increasing computational capabilities associated with embedded processing platforms, such platforms are of increasing interest for deploying signal processing systems with neural networks. However, unlike high-performance computing platforms, which are suitable for computationally-intensive neural-network-equipped systems, embedded platforms are often characterized by tight resource constraints. These constraints necessitate new types of optimization and trade-off analysis in the complex design spaces associated with neural network implementation. Resource constraints also become a major concern in the simulation of spiking neural networks (SNNs) on commodity-off-the-shelf (COTS) desktop or laptop computing platforms. Such simulation capability opens up much greater access to accurate SNN simulation, which is conventionally carried out on supercomputers or specialized hardware. This thesis focuses on developing novel models and methods

for efficient simulation and implementation of neural networks on resource-constrained platforms.

First, we present a novel approach for simulating Spiking Neural Networks (SNNs) that is based on timed dataflow graphs. Whereas conventional SNN simulators compute changes in spiking neuron variables at each time step, the proposed simulation approach focuses on evaluating spike timings. This focus on evaluating when a dataflow actor (spiking neuron) reaches a new spike contributes to making spike evaluation an event-driven computation. The resulting event-driven simulation approach avoids unnecessary computations at time steps that lie between spiking events. This optimization is achieved while avoiding the large overheads associated with lookup tables that are incurred in existing event-driven approaches. Our results show identical spiking behavior compared to simulation using a conventional (time-based) simulator while providing significant improvement in execution time. Furthermore, the simulation of the event-driven approach is achieved on a low cost, COTS computer, whereas most SNN simulators have focused on supercomputer scale platforms or specialized hardware, as described above.

Secondly, this thesis also investigates the implementation of deep neural networks (deep convolutional neural networks in particular) on resource-constrained platforms. This study is carried out in the context of hyperspectral image processing, which has attracted increasing research interest in recent years, due in part to the high spectral resolution of hyperspectral images together with the emergence of deep neural networks (DNNs) as a promising class of methods for analysis of hyperspectral images. An important challenge in realizing the full potential of hyperspectral imaging technology is the problem of deploying image analysis capabilities on resource-constrained platforms,

such as unmanned aerial vehicles (UAVs) and mobile computing platforms. In this thesis, we develop a novel approach for designing DNNs for hyperspectral image processing that are targeted to resource-constrained platforms. Our approach involves optimizing the design of a single DNN for operation across a variable number of spectral bands. DNNs that are developed in this way can then be adapted dynamically based on the availability of resources and real-time performance constraints. The proposed approach supports the Dynamic Data Driven Application Systems (DDDAS) paradigm as an integrated part of the design and training process to enable dynamic-data driven adaptation of the DNN structure — that is, the set of computational modules and connections that are active when the DNN operates. We demonstrate the effectiveness of the proposed class of adaptive and scalable DNNs through experiments using publicly available remote sensing datasets.

Deep Neural Networks (DNNs) are adopted in numerous application areas of signal and information processing with Convolutional Neural Networks (CNNs) being a particularly popular class of DNNs. Many machine learning (ML) frameworks have evolved for design and training of CNN models, and similarly, a wide variety of target platforms, ranging from mobile and resource-constrained platforms to desktop and more powerful platforms, are used to deploy CNN-equipped applications. To help designers navigate the complex design spaces involved in deploying CNN models derived from ML frameworks on alternative processing platforms, retargetable methods for implementing CNN models are of increasing interest.

In this thesis, we present a novel software tool, called the Lightweight-dataflow-based CNN Inference Package (LCIP), for retargetable, optimized CNN inference on different hardware platforms (e.g., x86 and ARM CPUs, and GPUs). In LCIP, source code

for CNN operators (convolution, pooling, etc.) derived from ML frameworks is wrapped within dataflow actors. The resulting coarse grain dataflow models are then optimized using the retargetable LCIP runtime engine, which employs higher-level dataflow analysis and orchestration that is complementary to the intra-operator performance optimizations provided by the ML framework and the back-end development tools of the target platform. Additionally, LCIP enables heterogeneous and distributed edge inference of CNNs by offloading part of the CNN to additional devices, such as onboard GPU or network devices. Our experimental results show that LCIP provides significant improvements in inference throughput on commonly-used CNN architectures, and the improvement is consistent across desktop and resource-constrained platforms.

Lastly, image classification is an essential challenge for many types of autonomous and smart systems. With advances in Convolutional Neural Networks (CNNs), the accuracy of image classification systems has been dramatically improved. However, due to the escalating complexity of state-of-the-art CNN solutions, significant challenges arise in implementing real-time image classification applications on resource-constrained platforms. The framework of elastic neural networks has been proposed to address trade-offs between classification accuracy and real-time performance by leveraging intermediate early-exits placed in deep CNNs and allowing systems to switch among multiple candidate outputs, while switching off inference layers that are not used by the selected output. In this thesis, we propose a novel approach for configuring early-exit points when converting a deep CNN into an elastic neural network. The proposed approach is designed to systematically optimize the quality and diversity of the alternative CNN operating points that are provided by the derived elastic networks. We demonstrate the utility of the pro-

posed elastic neural network approach on the CIFAR-100 dataset.

EFFICIENT SIMULATION AND IMPLEMENTATION OF NEURAL NETWORKS ON RESOURCE-CONSTRAINED PLATFORMS

by

Lei Pan

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in fulfillment
of the requirements for the degree of
Doctor of Philosophy
2023

Advisory Committee:
Professor Shuvra S. Bhattacharyya, Chair/Advisor
Professor Manoj Franklin
Professor Behtash Babadi
Professor Patrick McCluskey, Dean's Representative
Dr. Jerry Wu

To my parents for their support.

To Chunxing and Sassafras.

Acknowledgements

The completion of this dissertation would not have been possible without the support of many people.

First and foremost, I would like to express my deepest gratitude to my Ph.D. advisor, Professor Shuvra Bhattacharyya, for his invaluable support and guidance during my doctoral studies. His exceptional intellect, patience, and unwavering commitment to excellence in academia and teaching have been a constant source of inspiration for me. It is a great privilege to have had the opportunity to be mentored by him throughout my Ph.D. journey.

I also want to thank my committee members. I would like to thank Professor Manoj Franklin for being my academic advisor during my Master studies at the ECE department. Additionally, I would like to extend my appreciation to Doctor Jerry Wu for his continuous assistance and guidance during my transition from Master's to Ph.D. studies, both academically and professionally. I am also appreciative of Professor Behtash Babadi and Professor Patrick McCluskey for serving on my committee and their invaluable insights, suggestions, and comments.

I am grateful to Dr. François Christophe, Professor Tommi Mikkonen, Professor Zhu Li, Mr. Rijun Liao, Ms. Yan Zhang, Dr. Eung Joo Lee, Mr. Yi-Ting Shen, Mr. Yi Zhou, Dr. Heikki Huttunen, Dr. Honglei Li, Dr. Jiahao Wu, Dr. Yanzhou Liu, Ms. Yaesop Lee, Mr. Jing Xie, Dr. Xiaomin Wu, and other colleagues and research project collaborators for their generous assistance, valuable discussions, and collaboration.

I would also like to express my gratitude to Dr. Kazuo Nakajima, who introduced

me to the world of embedded systems and digital design and offered me a teaching assistantship position to his class. His passion for teaching had a profound influence on my graduate studies. I am also deeply thankful to Dr. Kofi Boahene, Dr. Christine Gourin, and Dr. Lauren Bolding from the Johns Hopkins Hospital for their exceptional medical expertise and surgical skills. Without them, I would not have had the opportunity to pursue my Ph.D. in the first place. Additionally, I want to extend my thanks to Dr. Zhongzheng Tian and Mr. Xiangxiang Kong for their friendship and support throughout my graduate studies.

Last but not least, I would like to give special thanks to my wife, Chunxing Yin, and my parents for the endless, unconditional love and support.

The research underlying this thesis was supported in part by the U.S. Air Force Office of Scientific Research under the DDIP Program, Army Research Office, and Army Research Laboratory (ARL).

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AdEx	Adaptive Exponential
AI	Artificial Intelligence
API	Application Programming Interface
ARM	Advanced RISC Machine
BN	Batch Normalization
CFDF	Core Functional Data Flow
CIFAR	Canadian Institute for Advanced Research
CNN	Convolutional Neural Network
COTS	Commodity-Off-The-Shelf
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DCI	Diversity Comparison Indicator
DCT	Discrete Cosine Transform
DCNN	Deep Convolutional Neural Network
DDDAS	Dynamic Data Driven Application Systems
DNN	Deep Neural Network
EDSR	Enhanced Deep Super-Resolution Networks
FC	Fully Connected
FEA	Finite Element Analysis
FIFO	First-In-First-Out
FLOP	Floating Point Operation
FPGA	Field-Programmable Gate Array
GB	Gigabyte
GDT	Global Dataflow Time
GFLOP	Giga Floating Point Operation
GPU	Graphics Processing Unit
HIC	Hyperspectral Image Classification
HDL	Hardware Description Language
HSI	Hyperspectral Image
HSIP	Hyperspectral Image Processing
Hz	Hertz
IAR	Incremental Actor Re-assignment
IBM	International Business Machines
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
kHz	Kilo-Hertz
LCIP	Lightweight-dataflow-based CNN Inference Package

LD	Lightweight Dataflow
LDSS	Lightweight Dataflow Spiking neural network Simulator
LIDE	Lightweight Dataflow Environment
LLCL	Low-Level Compute Libraries
LORSAL	Logistic Regression via Variable Splitting and Augmented Lagrangian
ML	Machine Learning
MPPA	Massively Parallel Processor Array
MPSoC	Multiprocessor systems-on-chip
ms	Milliseconds
NASA	The National Aeronautics and Space Administration
nm	Nanometer
pA	Pico-Amperes
PCA	Principle Component Analysis
PCNN	Pulse-Coupled Neural Networks
PSNR	Peak Signal-to-Noise Ratio
QNNPACK	Quantized Neural Networks PACKage
RAM	Random Access Memory
ReLu	Rectified Linear Unit
S.D.	Standard Deviation
SDF	SYnchronous Dataflow
SNN	Spiking Neural Network
SRCNN	Super-Resolution Convolutional Neural Network
SVM	Support Vector Machine
TPU	Tensor Processing Unit
TVM	Tensor Virtual Machine
UAV	Unmanned Aerial Vehicle
VBIC	Variable Band Image Classification
VDSR	Very Deep Super Resolution
VGG	Visual Geometry Group
Γ	Gamma
τLIDE	LIDE with extensions for timed dataflow modeling
Ω	Omega

Chapter 1

Introduction

Neural networks have been an important topic studied by many researchers as an alternative to other machine learning methods, largely due to the superior performance introduced by neural-network-based methods in many important application areas, such as application areas in computer vision. As neural networks become more widely adopted in many signal and information processing systems, the complexity of state-of-the-art neural network models also grows [1]. The increased complexity of neural networks has helped to drive advances in increasingly powerful processor technology, such as multicore processors, graphics processing units (GPUs), and tensor processing units (TPUs) [2], which can be used for more efficient execution of neural networks.

While much of the advances in processor technology for neural network implementation have been oriented towards high performance computer systems, there is also significant interest in processors that help to deploy neural network models at the network edge, where platforms typically have strict constraints in terms of processing resources, memory requirements and energy consumption. Compared to high-performance computing platforms, platforms for edge computing are more energy efficient and compact in size. The platforms are typically equipped with multicore CPUs and GPUs that are significantly smaller in scale and less powerful in terms of computational capability — compared to high-performance platforms — and much less demanding in terms of

power consumption. Edge-based platforms may also include hardware accelerators that are streamlined for resource-constrained machine learning.

Efficient operation under strict resource constraints is also relevant in simulation of neural networks — in particular, for simulation of spiking neural networks (SNNs) on commodity-off-the-shelf (COTS) desktop or laptop computing platforms. Conventional SNN simulators are targeted to supercomputers, very large scale computing servers, or expensive hardware accelerators. The ability to simulate SNNs on COTS computing platforms opens up much greater access to accurate SNN simulation, and has the potential to increase the variety of ways in which SNNs are studied and applied in neuroscience and neuromorphic computing.

Deploying neural networks on resource-constrained platforms, including both simulation and implementation of neural networks, involves complex design spaces associated with the mapping of the networks onto the platforms and the configuration of the networks. These design spaces must be navigated effectively to achieve high-quality trade-offs among key design objectives, including energy efficiency, execution time, and inference or simulation accuracy [3, 4].

In this thesis, we present four research contributions that pertain to the efficient simulation and implementation of neural networks on resource-constrained platforms: (1) a new approach to simulating spiking neural networks (SNNs) with timed dataflow graphs, including the development of a novel simulation tool called LDSS (Lightweight Dataflow SNN Simulator); (2) a method for dynamic, data-driven hyperspectral image classification on resource-constrained platforms; (3) LCIP, the Lightweight-dataflow-based CNN (convolutional neural network) Inference Package for retargetable, optimized CNN in-

ference on different hardware platforms; and (4) a novel approach for configuring and selecting early-exit points in a deep CNN modeled as an elastic neural network.

The first contribution focuses on developing a new simulation methodology for SNNs that is based on timed dataflow modeling. The new methodology enables efficient event-driven SNN simulation, and facilitates the simulation of complex SNNs on commercial-off-the-shelf (COTS) computing platforms.

The second contribution focuses on the efficient implementation of neural networks on resource-constrained platforms — in particular, on deploying neural networks on such platforms with adaptive runtime performance and topology. Resource-constrained, adaptive implementation is studied in the context of hyperspectral image classification, which is an area that has attracted great interest in recent years for new neural-network-based methods. The proposed system design for hyperspectral image classification allows for adaptation across different system throughput and accuracy targets, while being amenable to deployment on resource-constrained processing devices.

The third contribution centers around LCIP, which provides an integrated approach to resourceability, efficiency, retargetability, and configurability in edge-based CNN inference. LCIP helps to bridge the gap between dataflow modeling and machine learning (ML) inference frameworks, allowing flexible high-level graph scheduling on resource-constrained devices. LCIP adapts dataflow methods to different platforms and libraries based on designer specifications.

The fourth contribution helps system designers to explore multi-objective design spaces for deep CNN (DCNN) inference optimization. By equipping DCNNs with multiple exit points, elastic neural networks allow for alternative trade-offs among multiple

performance metrics. In this contribution, we build upon prior work on elastic neural networks. The key novelty in our contribution lies in systematically incorporating concepts of multi-objective optimization and Pareto diversity into system design for elastic neural networks.

The remainder of this thesis is organized as follows. Chapter 2 introduces the proposed LDSS software framework for efficient simulation of SNNs. Chapter 3 introduces our dynamic, data-driven hyperspectral image classification system targeted to resource-constrained platforms. Chapter 4 presents the the Lightweight-dataflow-based CNN Inference Package (LCIP), and Chapter 5 presents our multi-objective framework for selecting early exit points in elastic neural network structures for DCNNs. The contributions of the thesis are summarized and directions for future work are discussed in Chapter 6.

Chapter 2

LDSS for Simulating Spiking Neural Networks with Timed Dataflow Graphs

In this chapter, we focus on introducing LDSS, the Lightweight Dataflow SNN Simulator, a simulator designed specifically for efficient Spiking Neural Network (SNN) simulation. We begin by providing background information on SNN simulation and discussing two commonly used simulation approaches: time-driven and event-driven. We then delve into the design and development of LDSS, with a particular emphasis on its event-driven SNN simulation capabilities. Through a series of experiments, we demonstrate the effectiveness and efficiency of LDSS compared to a suitably-formulated time-driven baseline.

Material in this chapter was published in partial, preliminary form in [5].

2.1 Introduction

The simulation of biological computation with Spiking Neural Networks (SNNs) has recently gotten attention for saving energy during computation [6, 7]. SNNs are often used in computational neuroscience for modeling biological behavior comparable to in-vitro or in-vivo experiments. In recent years, SNNs have also been applied to circuits and systems for machine learning.

In this chapter, we propose a novel event-driven approach for simulating SNNs, and

we present a prototype simulator called LDSS (Lightweight Dataflow SNN Simulator), which demonstrates this approach. LDSS is developed using the Lightweight Dataflow Environment (LIDE) [8], which is a tool for dataflow-based design of signal processing systems.

The simulation approach used in LDSS involves a finite-element evaluation to predict the next time a given neuron fires a spike. This type of prediction is used to make the computation of spikings in the network event-driven instead of time-driven. Such event-driven computation is enabled by the design of SNNs as dataflow graphs, where each actor (vertex) of a graph represents a spiking neuron and encapsulates computation that predicts the next firing time of the neuron given its current state. In addition to improving simulation speed through event-driven simulation, the dataflow-based formulation of the proposed simulation approach facilitates retargetability across diverse hardware platforms, such as CPU, FPGA, or GPU platforms, or their hybrid combinations.

The rest of this chapter is organized as follows. After a brief state of the art on SNN simulation and simulators making usage of dataflow engines (Section 2.2), we present the background of neuron models and dataflow framework used for the realization of our simulator in Section 2.3. Implementation details of our simulator are described in Section 2.4 followed by a comparison with a sequential version of the same simulation written in C (Section 2.5). Section 2.7 provides our initial conclusion on developing a SNN simulator with a dataflow approach as well as our plan for future directions.

2.2 Related Work

Various SNN simulators exist (e.g., see NEURON [9], NEST [10], Brian [11, 12]). Each simulator has its own specificity and purpose, but generally, they have evolved into providing an abstraction layer for computational neuroscientists to focus mainly on the specifications of their simulations, may it be at the molecular level or for the description of a large population of highly interconnected neurons.

SNN simulators are developed following two different approaches: time-driven or event-driven simulation, with time-driven approaches being more common. The preference for time-driven approaches stems in part from the slower speed of existing event-driven SNN simulators. This slow speed results from extensive use in existing event-driven simulators of pre-compiled lookup tables [13], where the dimensionality of the tables and the frequency of their access become bottlenecks. As a result, time-driven approaches outperform conventional event-driven approaches for complex networks [13]. The main contribution of this chapter is to introduce a new approach to event-driven SNN simulation that is based on dataflow modeling, and event-prediction functions that are encapsulated within dataflow actors. The proposed approach is shown to outperform time-driven simulation, with the performance gap increasing as the size of the network increases. Rather than seeking to introduce a new tool in the large space of existing SNN tools, our primary contribution is to introduce new insight into the trade-off between event- and time-driven simulation, which may in turn contribute to the improvement of existing tools or development of new ones.

There are only few prior developments to our best knowledge that focus on design-

ing and simulating SNNs with dataflow techniques. The NeuroFlow framework, which uses a proprietary dataflow engine as the hardware platform, is a noticeable example following this direction [14]. In contrast to NeuroFlow, our work applies dataflow as a programming model while imposing no constraints on the hardware platform. This facilitates retargetability to diverse platforms (e.g., single-core, multicore, or FPGA platforms), as described in Section 2.1. Moreover, the approach we propose in this chapter is entirely based on event-driven computation, whereas the NeuroFlow design computes internal parameter variations of each neuron of the network using a time-driven approach.

2.3 Background

2.3.1 Spiking Neuron Model

For the development of LDSS, we apply Izhikevich's simple model of spiking neurons [15] to simulate neurons in networks that are represented in LDSS. Izhikevich's model is selected as a first model for incorporation into LDSS because the model combines biological realism with computational simplicity. However, the main ideas of LDSS can be applied to any model of choice, such as AdEx [16] or Leaky Integrate-and-Fire.

A synaptic connection is often modeled as a weight w_{ij} randomly generated between two neurons i and j with the connection going from neuron i (pre-synaptic) to neuron j (post-synaptic). When the pre-synaptic neuron is excitatory, the weight w_{ij} is normalized to the range $[0, 0.5]$. Similarly, if the pre-synaptic neuron is inhibitory, $w_{ij} \in [-1, 0]$.

A network composed of such random constant synaptic weights for all its synaptic connections defines the class of pulse-coupled neural networks (PCNNs). In PCNNs,

synaptic connections are fixed at the initialization of the network and weights remain constant throughout the simulation. The simulation experiments presented in this chapter are based on this type of synaptic connectivity in order to present the validity of our dataflow-based and event-driven simulation approach with comparison to a reference sequential, time-driven C-language version.

Other models of synaptic connections exhibiting short-term [17] and long-term plasticity [18] exist in the literature. These are not within the scope of this chapter and will be subjects of future work in the development of LDSS.

2.3.2 Dataflow Modeling

Dataflow-based design models of signal and information processing systems are directed graph models in which graph vertices, called actors, represent computational modules, and each edge represents the communication of data from the output of one module to the input of another. A recent study illustrating the utility of dataflow techniques in the context of AI circuits and systems is presented by Li et al. [19]. In LDSS, each actor corresponds to an individual neuron, and encapsulates the computations that are used to simulate the behavior of the neuron. A dataflow edge $e = (u, v)$ in LDSS correspond to synaptic connections between pairs of neurons with the source and sink vertices of e correspond to the pre-synaptic $\mu(u)$ and post-synaptic $\mu(v)$ neurons respectively. Here, for a given LDSS actor x, we use $\mu(x)$ to denote the neuron that is represented by x. For detailed background on dataflow-based modeling of signal and information processing systems, we refer the reader to [20, 21].

For implementation of dataflow graphs in LDSS, we employ the Lightweight Dataflow Environment (LIDE) [8]. LIDE includes application programming interfaces (APIs) for developing actors and edges in signal processing dataflow graphs. The APIs are not language-specific; instead, they are defined in terms of fundamental dataflow principles. In LDSS, we employ LIDE-C, which is based on C-language implementations of the LIDE APIs, and τLIDE, which extends LIDE-C with capabilities for timed simulation [22].

τLIDE [22] is a time-extended version of LIDE. While pure dataflow models of computation are untimed models with no built-in concept of time or time stamps, τLIDE incorporates the notion of a global time clock, and associates functions or constant values for estimating the time that is elapsed (within the "modeling universe" that is being simulated) when actors execute. Thus, τLIDE provides a tool for implementing and simulating dataflow-based system models in which the progression of time is an important aspect. The integrated support for time in τLIDE is important for simulating SNNs because of, for example, the dependence on time in Eq. 2.1 and Eq. 2.2 [15].

$$\begin{cases} \dfrac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \\ \dfrac{du}{dt} = a(bv - u) \end{cases} \tag{2.1}$$

$$\text{if } v \geq v_{peak} \text{ then, } \begin{cases} v = c \\ u = u + d \end{cases} \tag{2.2}$$

Here, the membrane potential v represents the voltage across the neuronal membrane, while the membrane recovery variable u interacts with v to model sub-threshold

dynamics. The membrane time constant a controls the decay rate of the membrane potential, determining how quickly the neuron returns to its resting potential after spiking. The membrane sensitivity b influences the responsiveness of the membrane potential to input currents. When a spike occurs, the membrane potential and recovery variable are reset to the values defined by the membrane reset potential c and recovery variable reset d, respectively. Lastly, the input current I represents the total input current received by the neuron, incorporating synaptic inputs, external inputs, and intrinsic currents.

2.4 Modeling and Simulation of SNNs in LDSS

In LDSS, each dataflow actor corresponds to an individual neuron, and encapsulates the computations that are used to simulate the behavior of the neuron. The actor models of neurons are based on Izhikevich's model, as implied in Section 2.3.1. Dataflow edges in LDSS correspond to synaptic connections between pairs of neurons. Given an edge e in an LDSS dataflow graph, the source and sink vertices of e correspond to the pre-synaptic and post-synaptic neurons associated with the connection modeled by e.

During the simulation of a system model in τLIDE, the current value of the global time clock of τLIDE is referred to as the global dataflow time (GDT). In co-simulation contexts where τLIDE is interfaced with one or more cooperating simulators, the GDT is easily distinguished from any clocks that are associated with those simulators. In LDSS, however, no cosimulation is involved since τLIDE is sufficient for implementing all of the required simulation functionality.

LDSS can be viewed as a package of SNN-specific actors and utilities that we have

developed for simulating and experimenting with SNNs in τLIDE. The utilities in LDSS include functions for automatically generating SNN models of arbitrary size using randomization techniques, and for reading SNN models from input files. This latter feature is useful to experiment with models that are generated by other SNN experimentation tools. LDSS can readily be extended with libraries that support other models for spiking neurons (beyond Izhikevich's model [15]).

The dataflow graph of an SNN model in LDSS generally consists of actors that represent both excitatory neurons and inhibitory neurons. Excitatory and inhibitory neurons in LDSS are modeled as instances of two different actor types. In the SNN models studied in our experiments, the ratio of excitatory neurons to inhibitory neurons is 4:1.

Fig. 2.1 illustrates an example of a simple dataflow graph representation of an SNN in LDSS. This example involves a network of five neurons, with 4 of them (E_1, E_2, E_3, E_4) being excitatory neurons and one of them (I_1) being an inhibitory neuron. The edges in Fig. 2.1 model synaptic connections between neurons. In particular, a dataflow edge (X, Y) in LDSS represents a synaptic connection between $\mu(X)$ and $\mu(Y)$, where $\mu(X)$ is the pre-synaptic neuron and $\mu(Y)$ is the post-synaptic neuron. Here, for a given LDSS actor X, we use $\mu(X)$ to denote the neuron that is represented by X. The numbers next to the edges shown in Fig. 2.1 show the weights of the corresponding synaptic connections (see Section 2.3.1).

Suppose that G is a dataflow graph that models an SNN in LDSS, and that N is an actor in G. The actor N has an associated execution time estimation function, which is denoted by θ_N, and computes the predicted time of the next spike that is to be generated by $\mu(N)$ based on the current state of $\mu(N)$. If no spiking event is predicted based on the

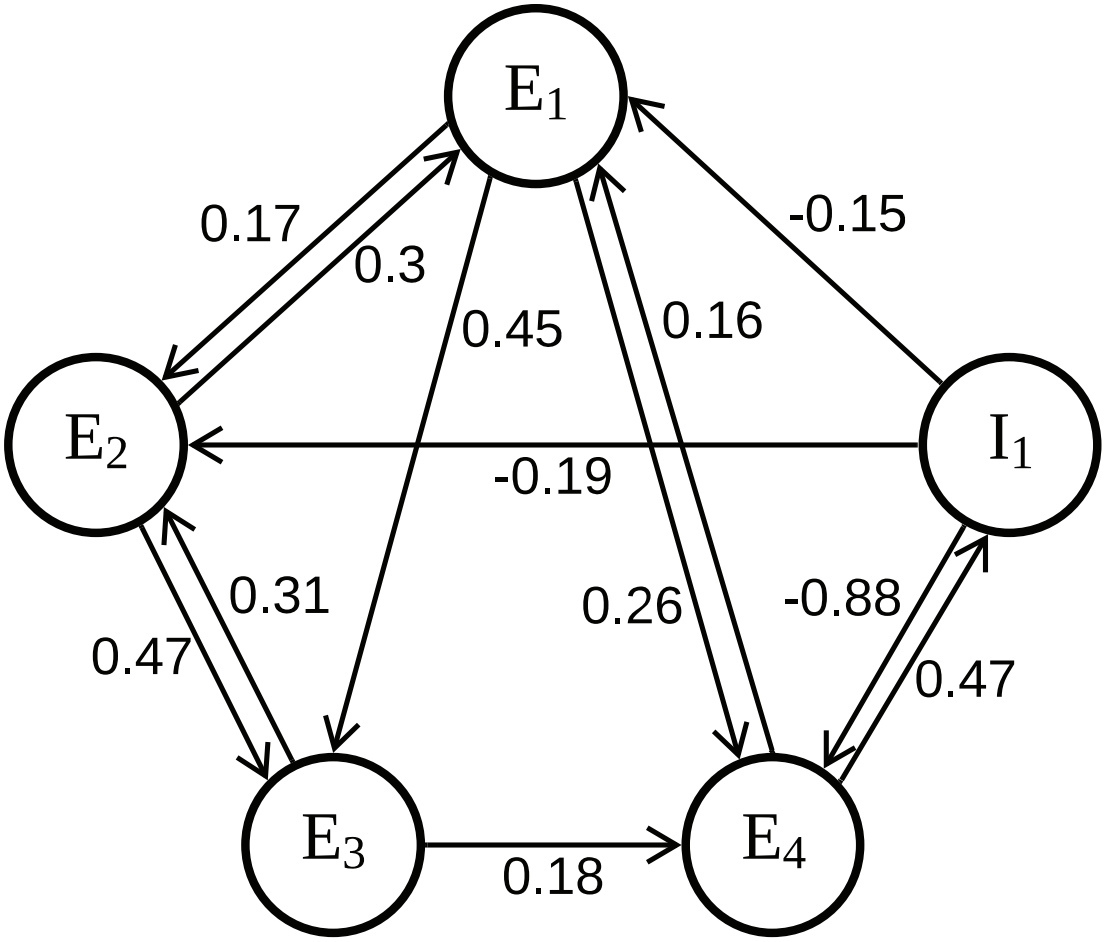


Figure 2.1: An example of an SNN dataflow graph in LDSS.

current neuron state, then θ_N returns ∞. This estimation function performs a finite element simulation based on Izhikevich's model whenever the actor N executes. The execution of N in turn is driven by the arrival of spikes at the inputs of N that have been generated from predecessor neurons. More specifically, the arrival of a spike at time t at any input of N initiates an execution of θ_N, which in turn may produce a predicted spiking time $p_N(t) > t$. The magnitude of current associated with a spike is represented as a dataflow token. In this context, the term "token" is the generic term for a quantum of information that is communicated across a dataflow edge.

Each synapse is modeled with a fixed propagation delay δ, which is the same for all synapses. Thus, if a neuron $\mu(A)$ generates a spike on dataflow edge e at time t_1, then the spike will arrive at the input of B at time $t_2 = t_1 + \delta$, where B is the sink actor of e. LDSS can readily be extended with more complex propagation delay models, which represents a useful direction for future work.

At any given value t_a of simulated time, a neuron $\mu(N)$ can have at most one predicted future spike that is pending to be processed by LDSS at some time $t_b > t_a$, where t_b has been determined by θ_N. Such a predicted spike is maintained as a pending event Y_b within the event list of LDSS. However, if $\mu(N)$ already has such a predicted future spike that is pending, and a new spike arrives on an input of N at some time t_c within the interval $[t_a, t_b)$, then the previously scheduled spiking event is de-scheduled (removed from the simulator event list), and (following the rules described above for initiating an execution of N), θ_N is invoked to determine whether or not a new spiking event is predicted, and to compute a new predicted spiking time $p_N(t_c)$ if such an event is predicted. If the invocation of θ_N results in a new predicted spiking time, then the new spiking event

is scheduled to occur at $p_N(t_c)$.

More specifically, if the incoming spike is from an excitatory neuron, then a new spiking event will be predicted, and based on properties of the underlying spiking neuron model, we will have $p_N(t_c) < t_b$. On the other hand, if the incoming spike is from an inhibitory neuron, then a new spike may or may not be predicted, depending on whether the neuron's membrane potential is lowered below v_{peak}. If a new spike is predicted, then the predicted time will satisfy $p_N(t_c) > t_b$.

Neuron $\mu(N)$ generates a spike when the membrane potential $v_{\mu(N)}$ reaches its peak value v_{peak}, as defined by Izhikevich's model [15]. The timing of this physical behavior is what is modeled by θ_N. When $\mu(N)$ generates a spike at some time t, the actor N produces a token that represents the spike on all of the m output edges of N in the dataflow graph.

2.5 Experiments

We evaluated LDSS by using it to model and simulate randomly-generated SNNs with varying numbers of neurons Q. Specifically, we experimented with 7 randomly-generated SNNs $\Gamma_1, \Gamma_2, \ldots, \Gamma_7$ containing Q = 1,000, 5,000, 10,000, 15,000, 20,000, 25,000, 30,000 neurons, respectively. For each of these SNNs, 80% of the neurons were randomly selected to be excitatory neurons, and the remaining 20% were inhibitory.

2.5.1 Random SNN Generation

We applied a random SNN generation approach proposed by Izhikevich [15]. In this approach, the edges and edge weights are derived from a randomly-generated $Q \times Q$

matrix W of floating point values. Each neuron μ has a unique index $I(\mu) \in \{1, 2, \ldots, Q\}$, which is used to index the rows and columns of W. Each network Γ_i is simulated 3 times using the baseline simulator (see Section 2.5.2), and simulated 3 times using LDSS with the exact same Γ_i.

The diagonal elements of W are all set to 0. Each off-diagonal element $W_{i,j}$ $(i \neq j)$ is derived from a uniform distribution within a finite interval $[w_1, w_2]$. The endpoints of this interval differ depending on whether n_i is excitatory or inhibitory, where n_i represents the neuron associated with neuron index i (that is, $i = I(n_i)$). If n_i is inhibitory, then we use $w_1 = -1, w_2 = 0$, otherwise, we use $w_1 = 0, w_2 = 0.5$. For each ordered pair (n_i, n_j) of distinct neurons, we interpret n_i to be a predecessor of n_j if $\text{abs}(W_{i,j}) \geq 0.1$, where abs represents the absolute value function. Furthermore, if n_i is a predecessor of n_j, then the weight of the corresponding synapse is set to $W_{i,j}$.

2.5.2 Baseline Simulator

As mentioned in Section 2.3, we evaluate LDSS by comparing its simulation output and execution time performance with a sequential, C-language implementation of SNN simulation based on Izhikevich's model [15], which is the same model used for LDSS, as described in Section 2.4. We refer to this sequential simulator implementation in the remainder of this chapter as the *baseline simulator*. As its name implies, we developed the baseline simulator to provide a reference point — using the same implementation language and spiking neuron model — for evaluating the dataflow-based and event-driven SNN simulation methods presented in this chapter.

The baseline simulator uses a conventional, time-driven simulator design. It does not employ dataflow concepts or event-driven simulation techniques, which are key features of LDSS. Our simulation experiments were designed carefully to use the same underlying neuron and synapse models so that the dataflow- and event-driven simulation approach of LDSS could be functionally validated, and its performance effects isolated. This experimental approach supports the primary aim of this chapter: to demonstrate the utility of the proposed new approach to integrating dataflow modeling and event-driven simulation for SNNs.

2.5.3 Functional Validation

To test the validity of simulation results produced by LDSS, we simulate each of the 7 randomly-generated, Q-neuron SNNs $\Gamma_1, \Gamma_2, \ldots, \Gamma_7$ using both the baseline simulator and LDSS. We simulate each network for 1,000 milliseconds (ms) of simulated time. The noise component of each neuron's background current is simulated as thalamic input, similarly as in [15]. The noise component is randomly updated at each simulated time step in the baseline simulator. In LDSS, the noise component values can be generated randomly or imported from a file. The latter option is useful when comparing output results with another simulator. For functional validation of LDSS, we import the noise components for the background currents of all neurons from a file that is exported from the baseline simulator to contain all of the randomly-generated noise component values. This allows both simulators to operate with exactly the same noise signals. For excitatory neurons, the background current noise component is uniformly distributed in the interval

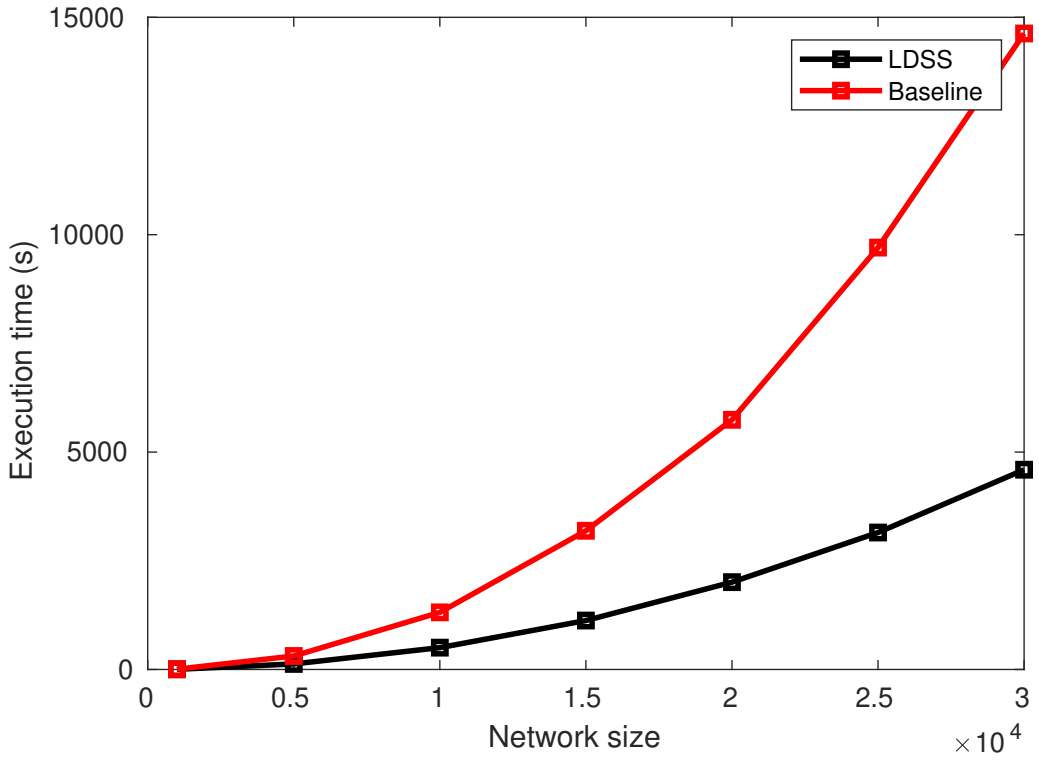


Figure 2.2: Execution time for variable network size Q.

[0,5] pico-Amperes (pA), and for inhibitory neurons, it is uniformly distributed in [0,4] pA.

We compare the simulated results in terms of the average spiking frequency over all of the neurons in a given network. In each comparison, we use the same network Γ_i for both the baseline simulator and LDSS. The results of these experiments show no discrepancy between the two simulators: for $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$, both simulators report the same average spiking frequencies — $(4.91\times10^{-3}, 5.22\times10^{-2}, 0.12, 0.14, 0.17, 0.21, 0.22)$, respectively. The units of the spiking frequency values reported here are kHz — e.g., a value of 0.12 corresponds to 120 Hz.

2.5.4 Performance Comparison

We also compared the execution time between LDSS and the baseline simulator, again using the same set of 7 randomly-generated SNNs. All simulations were executed on the same computer, which was equipped with an Intel Core i5-8259U CPU and 16GB of RAM. The simulations were each carried out through 1 second of simulated time. The execution time results are shown in Fig. 2.2. The horizontal axis corresponds to the network size Q, and the vertical axis corresponds to the measured execution time in seconds. The results illustrate that LDSS consistently outperforms the baseline simulator, with the amount of speedup increasing with the network size.

Detailed statistics associated with the results in Fig. 2.2 are shown in Table 2.1. Each row corresponds to one of the two simulators (Baseline or LDSS) applied to an SNN model with a given network size (Q). The values for the mean, minimum, maximum and standard deviation (S.D.) are all reported in seconds. The speed improvement over the 7 test networks ranges from 44.45% to 68.59% with LDSS performing increasingly better compared to the baseline as Q increases.

Moreover, the use of dataflow techniques in the design of LDSS facilitates acceleration using multicore processors or other types of parallel computing devices. Investigating additional performance improvements provided through such acceleration is a useful direction for future work.

We also compared the execution time between the baseline simulator and LDSS for a fixed network size and different amounts of simulated time. For this experiment, the network size was fixed at Q = 1,000, and the simulated time was varied across the

Table 2.1: Execution time comparison between the baseline simulator and LDSS for different network sizes.

Simulator, Netw. Size	Mean	Minimum	Maximum	S.D.
Baseline, $Q = 1,000$	6.13	6.06	6.27	0.12
LDSS, $Q = 1,000$	3.41	3.27	3.48	0.12
Baseline, $Q = 5,000$	310.04	302.86	317.09	7.11
LDSS, $Q = 5,000$	127.22	126.98	127.46	0.24
Baseline, $Q = 10,000$	1,313.36	1,285.01	1,335.58	25.84
LDSS, $Q = 10,000$	502.00	494.55	507.02	6.58
Baseline, $Q = 15,000$	3,190.28	3,184.49	3,193.27	5.02
LDSS, $Q = 15,000$	1,125.43	1,109.01	1,133.65	14.22
Baseline, $Q = 20,000$	5,742.75	5,654.09	5,889.29	127.84
LDSS, $Q = 20,000$	2,008.45	1,977.66	2,023.89	26.67
Baseline, $Q = 25,000$	9,703.44	9,617.40	9,752.64	74.77
LDSS, $Q = 25,000$	3,149.80	3,104.47	3,173.70	39.27
Baseline, $Q = 30,000$	14,628.38	14,505.39	14,721.99	111.24
LDSS, $Q = 30,000$	4,594.30	4,528.15	4,631.99	57.47

sequence $T = (1{,}000, 2{,}000, \ldots, 30{,}000)$, with units in milliseconds (ms). The resulting execution time comparison is shown in Fig. 2.3. Each experiment is repeated 3 times, and the reported result is the average execution time across the 3 repetitions.

The maximum standard deviation for the baseline simulator is 7.09 and the minimum standard deviation is 0.31. For LDSS, the maximum and minimum standard deviations are 4.69 and 0.09, respectively.

The results from Fig. 2.3 show that the relationship between execution time and simulated time is close to linear for both the baseline and LDSS simulators, with LDSS outperforming the baseline simulator by a minimum of 16.16% and maximum of 39.32% in terms of execution time.

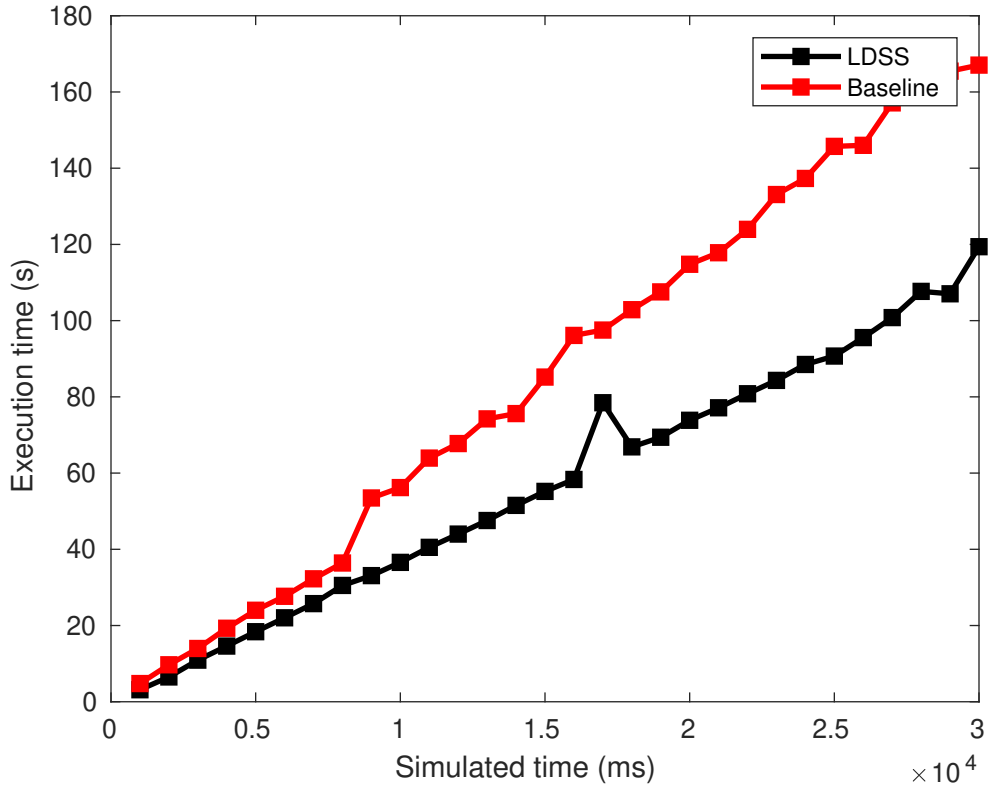


Figure 2.3: Performance comparison for different amounts of simulated time.

2.6 Additional Results

In this section, we present additional experimental results to provide more context into our comparison between LDSS and the baseline time-driven simulator.

The results presented in this section pertain to an experiment in which we generated neuron-spiking raster plots using LDSS and the baseline simulator. Figure. 2.4 and Figure. 2.5 summarize results from this experiment for the network having size $Q = 1{,}000$ and simulated time between 1,000 and 2,000 milliseconds (ms). In particular, Figure. 2.4 shows the neuron spiking raster plot obtained from the baseline simulator, and Figure. 2.5 shows the raster plot generated by LDSS. Each + in the raster plots represents the firing of a specific neuron (identified by the y coordinate) at a specific time (identified by the x coordinate).

2.7 Summary

In this chapter, we have presented a dataflow-based, event-driven approach for modeling and simulating spiking neural networks (SNNs), and we have prototyped this approach by developing a new simulation tool called the Lightweight Dataflow SNN Simulator (LDSS). Useful features of the proposed new simulation approach include improved simulation speed, enabled by the event-driven operation; design flexibility in that the simulation is not limited to a specific spiking neuron model or network topology; and retargetability across different hardware platforms and implementation languages, enabled by the rigorous use of dataflow modeling concepts. Interesting directions for future work include developing multi-threaded or GPU-accelerated versions of LDSS for further sim-

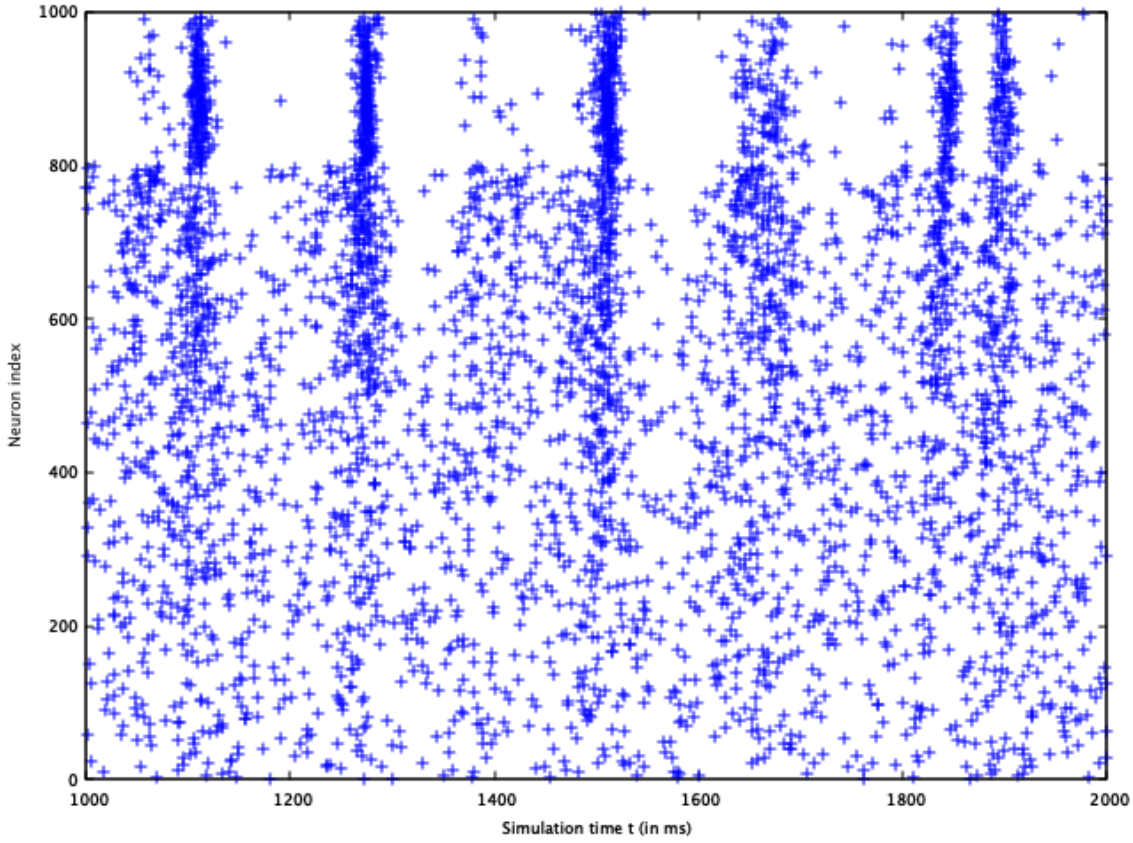


Figure 2.4: Neuron-spiking raster plot resulting from the baseline simulator $Q = 1000$.

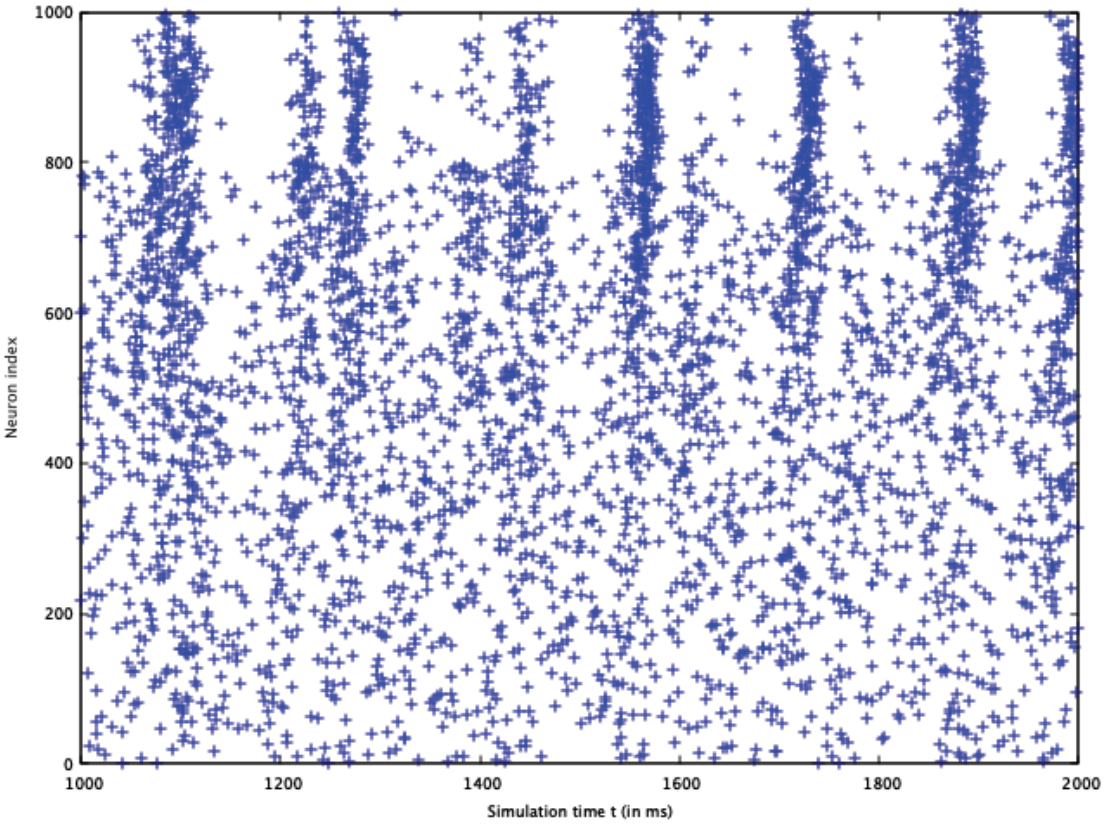


Figure 2.5: Neuron-spiking raster plot resulting from LDSS with $Q = 1000$.

ulation speedup, and adaptation of other SNN simulators using models and methods from the proposed simulation approach.

Chapter 3

Dynamic, Data-Driven Hyperspectral Image Classification on Resource-Constrained Platforms

In Chapter 2, we explored LDSS, an efficient Spiking Neural Network (SNN) simulator. In this chapter, our focus shifts to introducing a novel method for resource-constrained image-analysis implementation known as VBIC (Variable Band Image Classification). VBIC addresses the challenge of high complexity present in Hyperspectral Images (HSI), particularly in the spectral dimension, as well as the neural networks employed for classification purposes. The implementation of VBIC offers dynamic configurability within the Convolutional Neural Network (CNN) topology, enabling support for HSI inputs with varying degrees of spectral complexity.

Material in this chapter was published in partial, preliminary form in [23].

3.1 Introduction

Hyperspectral image processing (HSIP) applications are becoming increasingly common in important application areas such as surveillance [24], medical diagnostics [25], forensics [26], and remote sensing [27]. The utility of HSIP in these fields stems from the high levels of spectral diversity and spectral resolution that hyperspectral images provide compared to conventional image acquisition approaches (e.g., see [28]). An HSIP application of fundamental importance is the problem of *image classification*,

which involves mapping each pixel into a set of pre-determined classes. In recent years, deep neural networks (DNNs) have been shown to be useful for accurate image classification in HSIP applications [29]. Prior work on HSIP system design has focused on systems with loose resource constraints or without strong emphasis on data-driven adaptivity (e.g., see [30] [31]). However, the computational complexity of DNNs together with the large amounts of data involved in HSIP applications makes the deployment of DNNs in resource-constrained scenarios a challenging problem.

The capability for data-driven, real-time processing of hyperspectral image classification on resource-constrained platforms opens up the potential for many novel applications. In this chapter, we introduce a novel framework for hyperspectral image classification that integrates adaptive DNN-based image analysis with real-time, resource-constrained processing.

Our approach involves optimizing the design of a single DNN for operation across a variable number of spectral bands. DNNs that are developed in this way can then be adapted dynamically based on the availability of resources and time-varying constraints on real-time performance. The proposed approach allows the deployed DNN configuration to be varied at run time to maximize hyperspectral image analysis accuracy subject to operational requirements that may vary dynamically, and may be unknown at design time. We demonstrate the effectiveness of the proposed class of adaptive and scalable DNNs through experiments using publicly available remote sensing datasets.

This work helps to advance the application of Dynamic Data Driven Applications Systems (DDDAS) by providing new methods for encapsulating a range of HSIP configurations, with alternative trade-offs between complexity and image analysis accuracy,

within a single DNN model. The single model can be deployed onto a resource constrained platform and used to adapt system operation based on dynamically changing operational requirements — e.g., based on changes in urgency due to information extracted from recently-acquired imaging data.

3.2 Related Work

Hyperspectral image classification (HIC) is an important application in HSIP. It is applied, for example, in the area of remote sensing, where different types of land features need to be recognized and categorized. Most studies on HIC focus on maximizing classification accuracy. A common theme in recent works on HIC is the application of DNNs. Many of these recent works have reported very high accuracy when applying DNNs to HIC problems in remote sensing (e.g., see [32, 33, 34, 35]). However, these works often focus on accuracy without taking into account stringent resource constraints or real-time performance.

On the other hand, a number of studies have investigated real-time HIC. For example, Madroñal et al. developed a real-time HIC implementation on a high-performance computing platform called the Massively Parallel Processor Array (MPPA) [30]. Their approach utilized Support Vector Machine (SVM) methods for the classification process. Wu et al. proposed an approach called logistic regression via variable splitting and augmented Lagrangian (LORSAL) for GPU-based, real-time HIC [36]. Sharma et al. proposed a real-time DNN-based approach for face recognition from hyperspectral images [37].

Compared to the related work summarized above, the HIC method presented in this chapter is novel in its joint support for resource-constrained deployment, and scalable execution. Moreover our framework is based on DNN techniques, which have potential for very high accuracy HIC. Here, by *scalable*, we mean that trade-offs among accuracy, resource requirements, and real-time performance can be adapted flexibly and efficiently at run-time to match the configuration of the system to time-varying operational requirements.

3.3 Approach

The proposed system for HIC is designed to adaptively configure system complexity to maximize classification accuracy subject to constraints on real-time performance. The system is designed using a convolutional neural network (CNN) structure that accepts as input a variable number of hyperspectral input channels from among the complete set of channels S that is available from a given hyperspectral sensing subsystem. The elements of S are provided as input to the HIC system as an ordered list of channels $S = \{C_1, C_2, \ldots, C_m\}$, where m is the total number of available channels. The variable number of channels to use at run-time is selected from a set of predefined options $\Omega = \{n_1, n_2, \ldots, n_k\}$, where $k \in \{1, 2, \ldots, m\}$, and $1 \leq n_1 < n_2 < \ldots < n_k \leq m$. In our experiments, we use $k = 4$, $n_1 = 30$, $n_2 = 60$, $n_3 = 90$, and $n_4 = m$.

At run-time (during inference), an integer-valued input $n_c \in \Omega$ is provided to the HIC system to indicate which band-subset option is selected for image classification. The value of n_c gives the number of input channels that is to be used to classify image pixels;

the value of n_c can be varied dynamically by the system in which the CNN is embedded. More specifically, the set of channels used for classification is $\kappa = \{C_i \mid 1 \leq i \leq n_c\}$.

By definition of κ, the ordering $C_1, C_2, \ldots, C_m$ can be viewed as a priority list with lower-index channels having higher priority for inclusion in the inference process compared to higher-index channels. The priority list can be constructed using arbitrary methods for prioritizing hyperspectral imaging channels; in the experiments developed in this chapter, we apply the prioritization methods developed by Li et al. The prioritization methods were developed initially for multispectral video [38], and then extended to hyperspectral video [39].

We refer to our proposed approach as Variable Band Image Classification (VBIC). Our development of VBIC builds upon LDspectral, which was originally developed as a software tool for design optimization of dynamic, data-driven multispectral image processing systems [40], and has been extended more recently with support for hyperspectral image processing [39]. LDspectral in turn applies Lightweight Dataflow (LD), which is a compact set of application programming interfaces and associated libraries for dataflow-based design and implementation of embedded signal and information processing systems [41].

The CNN architecture for VBIC is illustrated in Figure. 3.1. The network is based on an HIC network proposed in [35]. In this work, we adapt the network of [35] to incorporate the novel capability of being able to operate on a variable number of spectral bands. We would like to emphasize that the adaptation approach that we develop in this chapter is not specific to the network of [35]. Instead, our approach can be viewed as a general methodology for adapting fixed-band-set CNNs for HIC into variable-band-subset

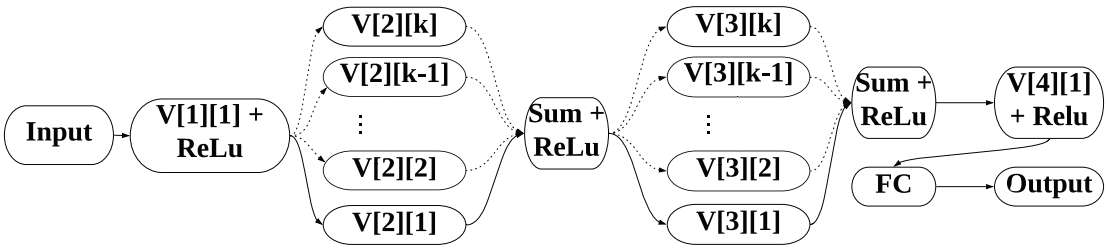


Figure 3.1: An illustration of the CNN architecture for VBIC.

form.

In Figure. 3.1, each block whose label starts with "V" represents a convolutional block. More precisely, each $V[a][b]$ represents $(2k + 2)$ convolutional blocks in the network performs a 3-D convolution. Each block labeled `ReLu` is a rectified linear unit, each `Sum` block performs the addition of results from the previous network stage, and the `FC` block is a fully connected layer. The dotted edges in Figure. 3.1 (e.g., the edge from the first ReLu block to $V[2][2]$) are connections that may or may not be active at run-time depending on how many bands the network is configured to execute (i.e., depending on the value of n_c) at that time. We refer to these connections as *dynamic connections*.

We refer to the subsystem consisting of $V[2][1], V[2][2], \ldots, V[2][k]$ as the *first stacked layer*, and similarly, we refer to the subsystem consisting of $V[3][1]$, $V[3][2]$, ..., $V[3][k]$ as the *second stacked layer*. For integers $x \in \{2, 3\}$, and $y \in [1, k]$, we use a minor abuse of notation to denote the set of convolutional blocks $\{V[x][1], V[x][2], \ldots, V[x][y]\}$ by $V[x][1 : y]$.

The input to the VBIC network is a three-dimensional tensor $T(p)$ with size $M \times M \times n_c$. The tensor $T(p)$ is used to classify a single image pixel p from a given hyperspectral image frame H. The tensor is referred to as the *patch* associated with p. The parameter

M is an odd integer that is fixed at design time, and is less than (typically much less than) the number of rows and number of columns in a single hyperspectral image frame. In our experiments, we use $M = 7$. The patch associated with p consists of all of the n_c selected spectral bands for pixel p and its neighbors within the $M \times M$ window within H that is centered at p. If p is at or sufficiently close to the boundary of H, the patch is zero-padded to produce a tensor of the required size $M \times M \times n_c$. A complete image frame H is classified by iteratively invoking the VBIC network on $T(p)$ for all pixels p in H. The output of the VBIC network for a given tensor $T(p)$ is a classification label for p together with a probability value, which indicates the level of confidence in the classification result.

Unique features of VBIC include: (1) n_c, the size of the third dimension of $T(p)$, can be varied dynamically, and (2) the network is trained deliberately to handle such a variable number of spectral bands in the classifier input.

We have developed a novel algorithm for VBIC training. The algorithm is shown in Algorithm 1, which provides a pseudocode sketch of the training process for VBIC. The function *initialTraining* takes as arguments an untrained VBIC network Γ with the structure illustrated in Figure. 3.1, a positive integer n, a labeled training dataset T, and a number of epochs E for training. The function configures Γ by deactivating all of the dynamic connections so that the only blocks $V[2][1]$ and $V[3][1]$ are active in the two stacked layers. The remaining stacked layer blocks are ignored in the training process for this function. The resulting configuration of Γ is then trained for E epochs using T. Only the first n channels (the highest priority n channels) of T are used in the training process.

The function *furtherTraining* takes as arguments a partially-trained VBIC network

Algorithm 1: A pseudocode sketch of the training process for VBIC.

input : Γ: A DNN network of the form illustrated in Figure. 3.1.
input : k: The number of band-subset options.
input : $\Omega = \{n_1, n_2, \ldots, n_k\}$: the number of bands in each band-subset option.
input : T: the labeled hyperspectral image dataset that is to be used for training.
input : $\{C_1, C_2, \ldots, C_m\}$: the priority list of spectral bands.
param: E: the number of training epochs in each training iteration.
output: $Weights_\Gamma$: the weights of the network Γ are trained as a side effect of this function.

```
Procedure VBIC-training Γ, k, Ω, T, {C_1, C_2, ..., C_m}
    channelCount = n_1;
    initialTraining(Γ, channelCount, T, E);
    for i = 2, 3, ..., k do
        channelCount = channelCount + n_i;
        furtherTraining(Γ, i, channelCount, T, E);
    end
    return Weights_Γ
end
```

Γ that has been produced by the previous iteration of the overall VBIC training process. The function arguments also include an integer $i \in [2, k]$, an integer $n \in [1, m]$ (recall that m is the total number of available hyperspectral channels), and as used in function *initialTraining*, a training dataset and number of epochs specification.

The function *furtherTraining* configures Γ by activating the stacked layer blocks $V[2][1 : i]$ and $V[3][1 : i]$ while deactivating all of the other stacked layer blocks. The function activates only those dynamic connections that are incident to activated blocks. In the resulting network, the stacked layer blocks $V[2][1 : i - 1]$ and $V[3][1 : i - 1]$ have weights that have been trained from preceding iterations of the enclosing training process. These existing weights are "frozen" along with the existing (pre-trained) weights of $V[4][1]$. Thus, the training process of function *furtherTraining* is configured to train only the weights of $V[1][1]$, $V[2][i]$, $V[3][i]$, and the fully connected layer `FC`. Based on

this configuration of frozen and non-frozen (to-be-trained) weights, the function carries out a training process with the given number of epochs and given training dataset. Only the first n channels of T (as specified by the third function argument) are used in the training process. The network Γ — in particular, its set of trained weights — is updated as a side effect of this function.

An important aspect of Algorithm 1 is the freezing of weights in stacked layer blocks $V[2][1:i-1]$ and $V[3][1:i-1]$ in each iteration $i = 2, 3, \ldots$ of the `for` loop, as enforced by function *furtherTraining*. This enables evolution of a single network that is capable of handling all of the pre-defined band subset options. Encapsulation of all of the options within a single DNN model is especially important in resource-constrained deployment scenarios, where there may be insufficient storage space available for multiple DNN models.

As the band subset option index j increases from 1 to k, the system accuracy can be expected to increase, while the processing complexity (and hence the execution time and energy consumption) also increases. The novel approach to designing and training the VBIC system provides systematic optimization of this trade-off between image classification accuracy and processing complexity, and encapsulation of the result compactly within a single DNN model.

3.4 Experiments

We demonstrate the proposed VBIC approach through experiments on an Android mobile phone (OnePlus 7 pro), which we use as a platform for prototyping resource-

constrained image processing applications. In the experiments, we provide hyperspectral image input to the platform through flash storage. The platform is equipped with an 8-core Qualcomm Snapdragon 855 CPU, 12 GB of RAM, and 256 GB storage.

We evaluate our VBIC-based Android implementation using two commonly used datasets in remote sensing: Indian Pines and Pavia University. The Indian Pines dataset has 145×145 pixels and 224 spectral bands with wavelengths ranging from 400–2500 nanometers(nm) [42]. The pixels are categorized into 16 classes. The Pavia University dataset has 610×610 pixels classified into 9 classes, and 103 spectral bands with spectral coverage spanning 430–860 nm.

The training process for VBIC (Algorithm 1) is implemented using PyTorch. Network training is performed using patch size parameter $M = 7$ (see Section 3.3), batch size of 40, learning rate of 0.1, weight decay of 0.01 for all layers, and number of epochs $E = 100$. The optimizer employed is AdaGrad [43]. Each of the two datasets investigated is partitioned into a training set and testing set. Each training and testing set contains 80% and 20% of the pixels of the associated dataset, respectively. Moreover, the partitioning is performed so that for each pixel class, 80% of the pixels in that class are in the training set and the other 20% are in the testing set. As stated in Section 3.3, we use $k = 4$, $n_1 = 30$, $n_2 = 60$, $n_3 = 90$, and $n_4 = m$ in our experiments.

Figure 3.2, 3.3, 3.4, and 3.5 illustrate the model size and overall accuracy for different n_c under two datasets used. Figure 3.2, 3.3, 3.4, and 3.5 show the model size and overall accuracy (testing accuracy across all pixel classes) for the VBIC system under the four different values for $n_c \in \Omega$. Here, by the *model size*, we mean the number of trainable parameters that is required in the network (excluding any parameters associated with

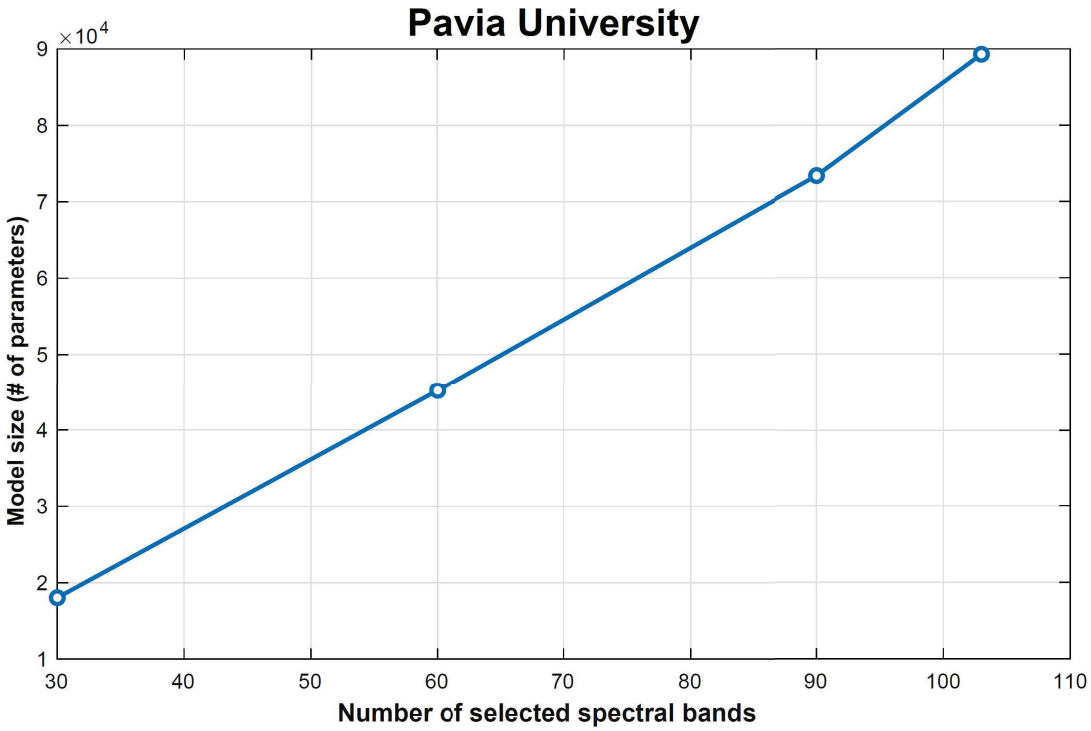


Figure 3.2: Model size for VBIC under different number of n_c with Pavia University dataset.

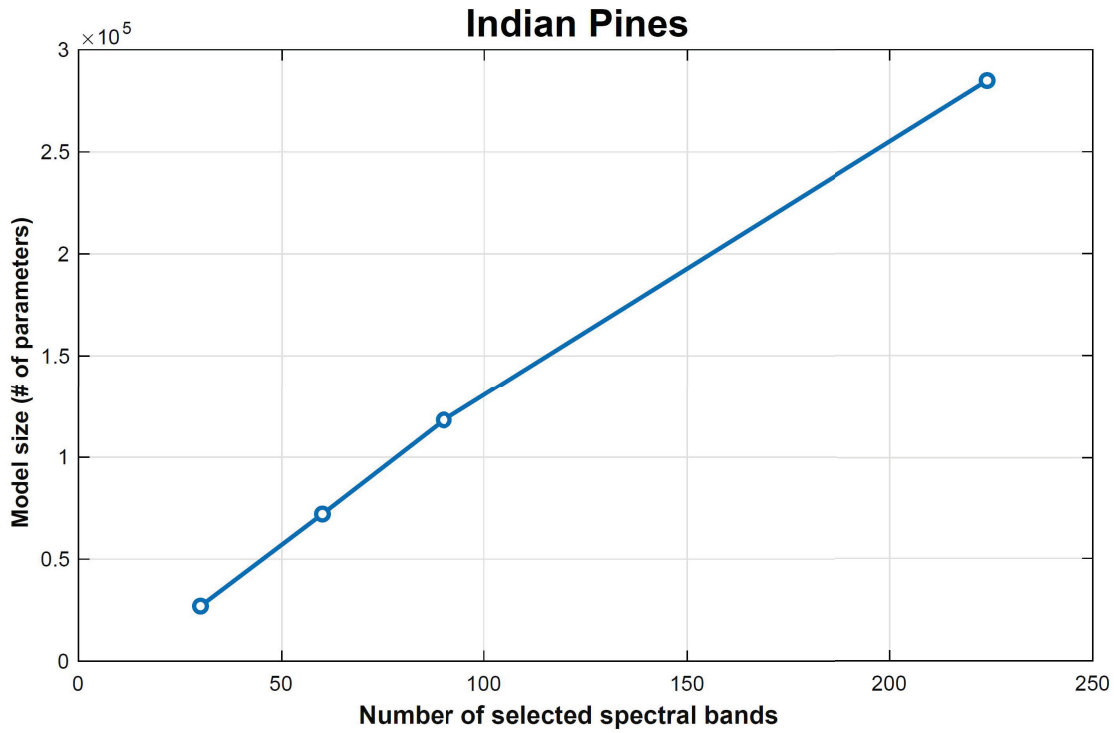


Figure 3.3: Model size for VBIC under different number of n_c with Indian Pines dataset.

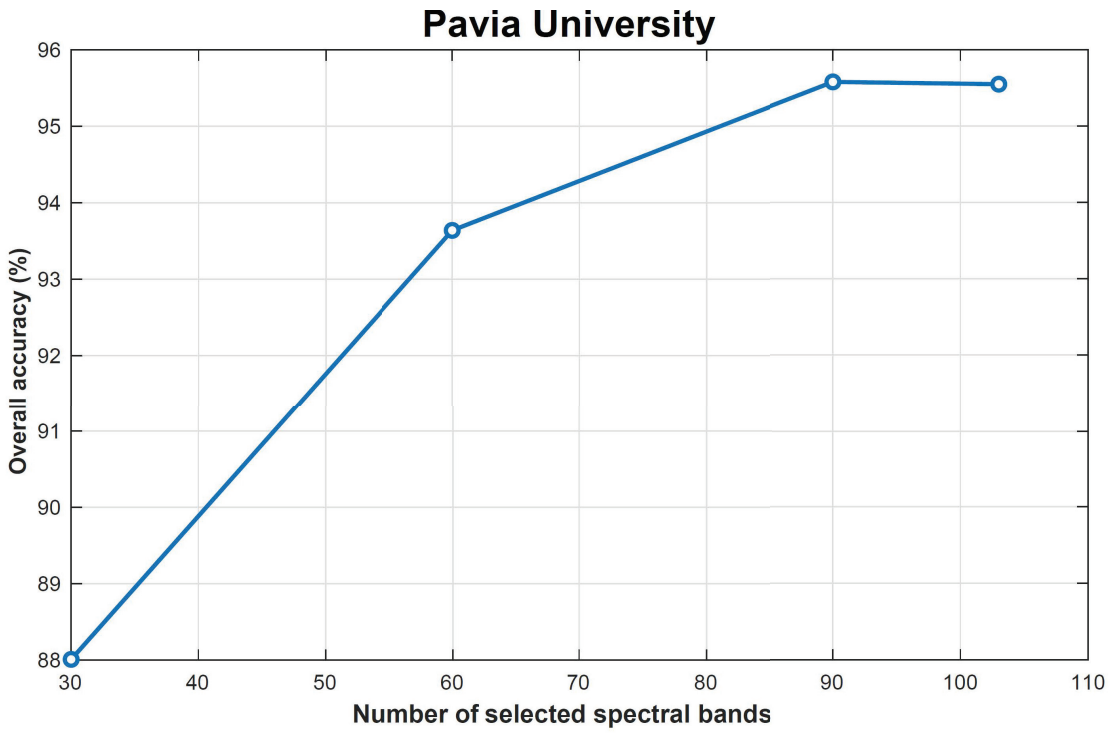


Figure 3.4: Overall accuracy comparison for VBIC under different number of n_c with Pavia University dataset.

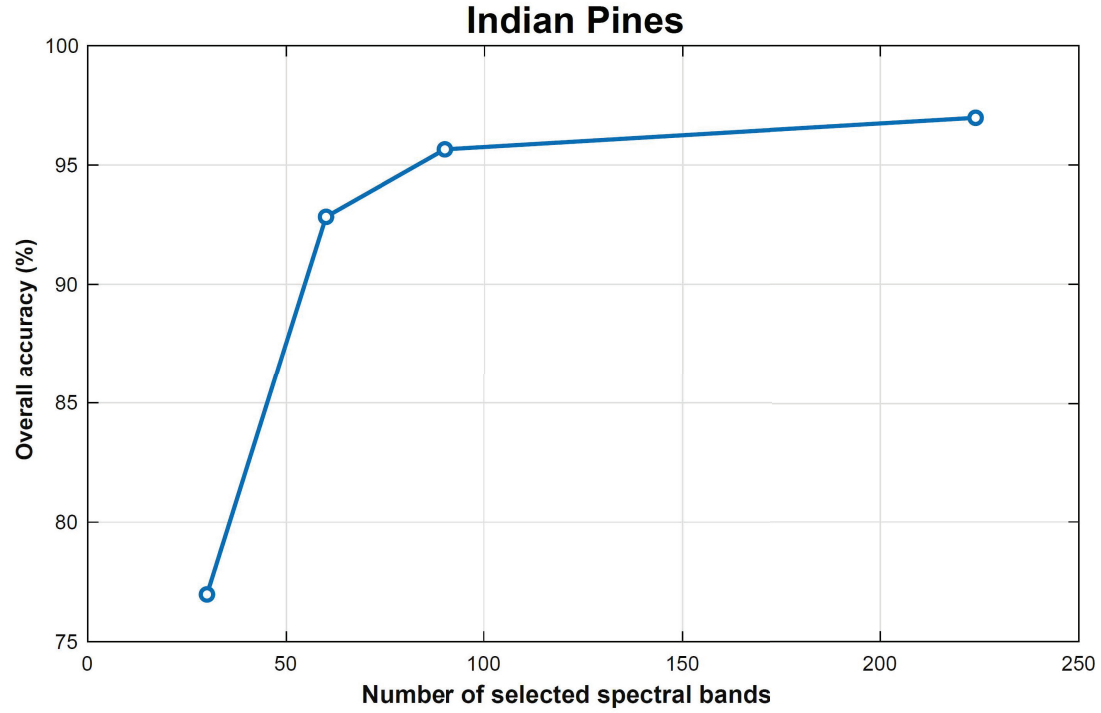


Figure 3.5: Overall accuracy comparison for VBIC under different number of n_c with Indian Pines dataset.

non-activated blocks). Note that only the model size associated with n_4 is relevant in assessing the overall model size since all of the models are supported in the VBIC system. However, the model sizes for different values of n_c provide insight into the underlying range of trade-offs provided between model complexity and accuracy. The row labeled "Maximum" represents $n_c = m$, where $m = 224$ and $m = 103$ for the Indian Pines and Pavia University datasets, respectively.

Table 3.1: Model size and accuracy.

	Pavia University Dataset		Indian Pines Dataset	
n_c	Model size	Overall accuracy	Model size	Overall accuracy
30	18,042	88.01%	27,009	76.98%
60	45,210	93.64%	72,097	92.83%
90	73,402	95.58%	118,209	95.66%
Maximum	89,306	95.55%	284,897	96.88%

We also measure the processing throughput and peak memory consumption (peak mem) of the VBIC system as it performs classification. The results are shown in Table 3.2. Each throughput value given in the table is derived by averaging over 20 repetitions of an experiment with the associated dataset and n_c value. The standard deviations computed for these 20 trials are listed in the column labeled "Std dev". The units for throughput are pixel classifications per second (PC/s). As shown in both Figure 3.6 and 3.7, the throughput of VBIC decreases as n_c increases, and the peak memory usage increases as n_c increases as in Figure 3.8 and 3.9, accordingly.

Table 3.2: Processing throughput and peak memory consumption.

	Pavia University Dataset			Indian Pines Dataset		
n_c	Throughput (PC/s)	Std dev (PC/s)	Peak mem (MB)	Throughput (PC/s)	Std dev (PC/s)	Peak mem (MB)
30	35,384.86	360.64	219	34,408.78	3,178.63	181
60	21,093.52	7.57	238	19,588.24	331.73	198
90	10,112.62	121.03	291	9,660.84	77.42	254
Maximum	6,888.42	47.04	374	4,803.96	102.41	338

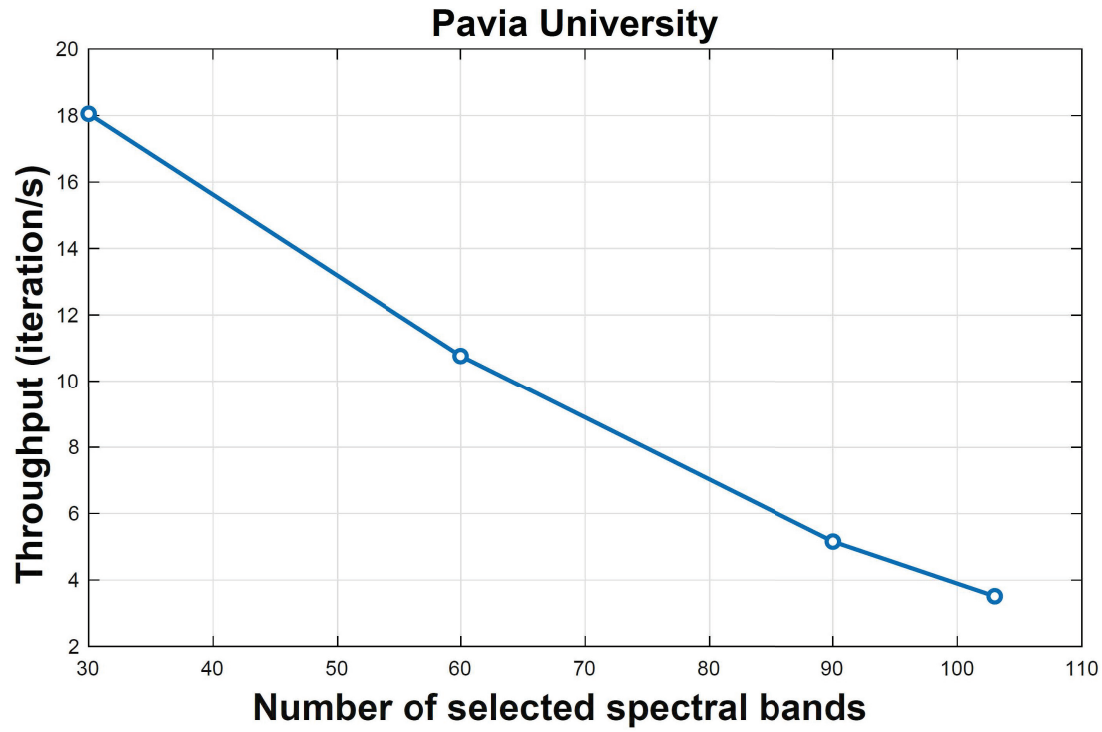


Figure 3.6: Throughput comparison for VBIC under different number of n_c with Pavia University dataset.

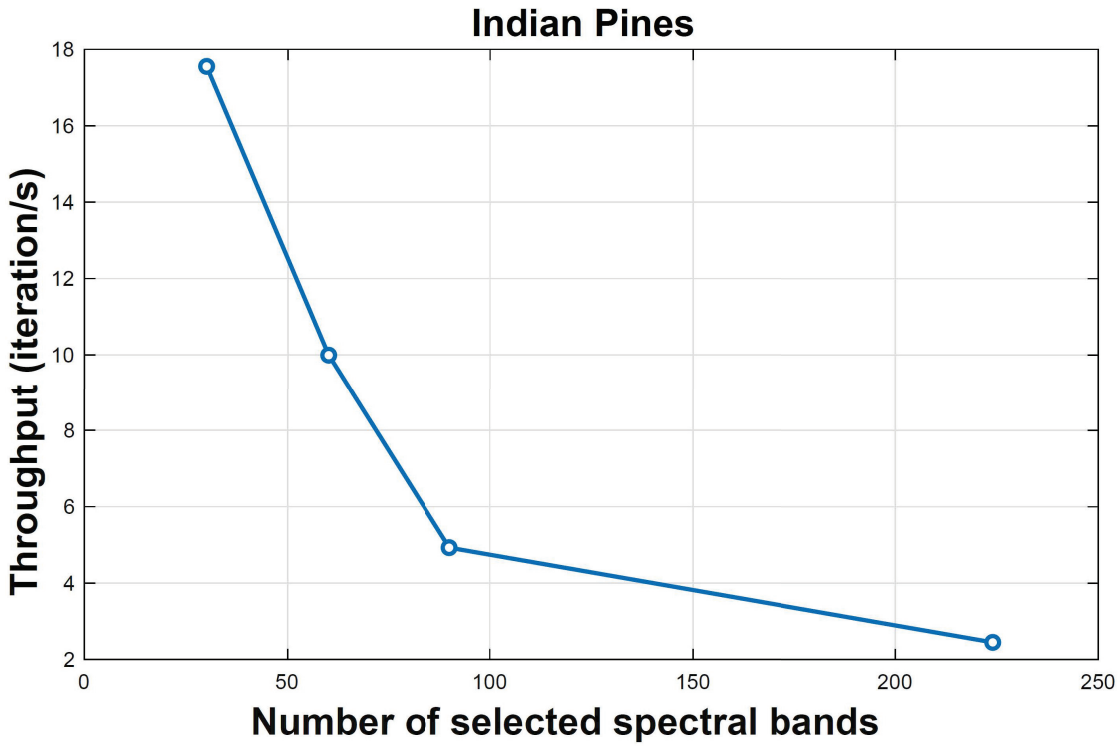


Figure 3.7: Throughput comparison for VBIC under different number of n_c with Indian Pines dataset.

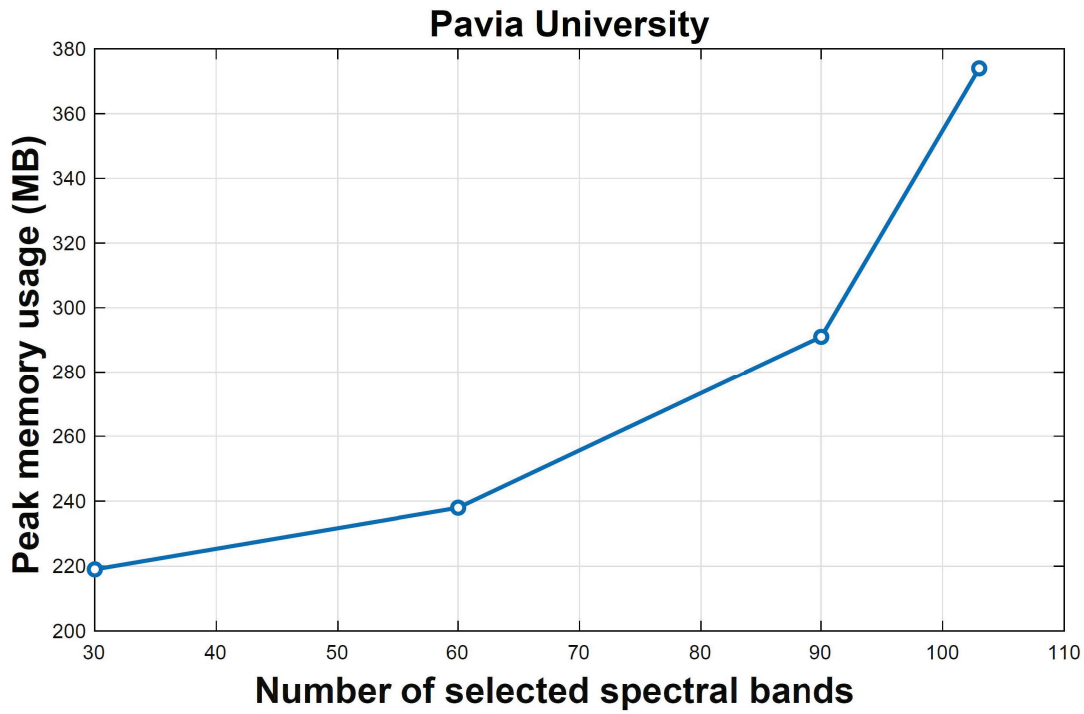


Figure 3.8: Peak memory usage comparison for VBIC under different number of n_c with Pavia University dataset.

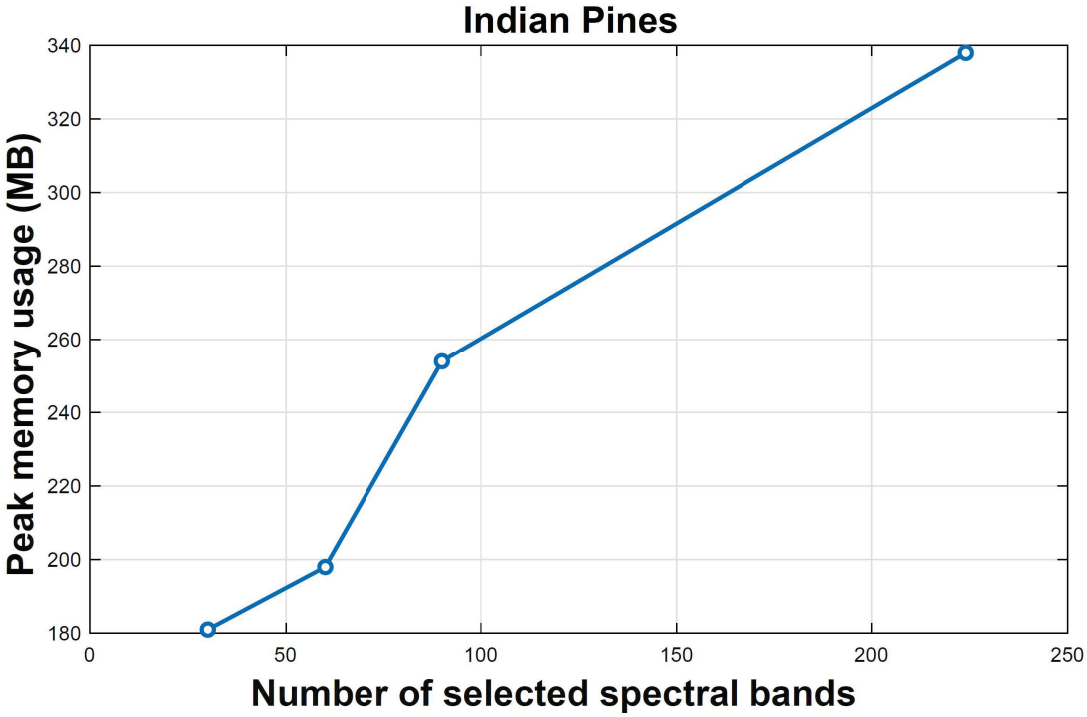


Figure 3.9: Peak memory usage comparison for VBIC under different number of n_c with Indian Pines dataset.

Together, the results in Table 3.1 and Table 3.2 demonstrate the effectiveness of the VBIC approach in practical classification scenarios. The results provide an example of the novel range of performance/accuracy trade-offs that can be deployed using VBIC. Moreover, the results demonstrate that large increases in throughput can be achieved with relatively small reduction in accuracy.

3.5 Additional Results

In this section, we provide additional experimental results to provide more context into the effectiveness of VBIC on a wider range of HSI datasets.

3.5.1 HSI Datasets

VBIC framework is further evaluated using commonly available public datasets. These datasets include: Indian Pines [42], Pavia University, Pavia Center, and Botswana. For all the datasets listed here, we use 80% of the pixels for training, and the remaining 20% for testing. The split is performed so that for each class, 80% of the pixels are in the training dataset and the remaining 20% are in the testing dataset. Among the pixels in the training dataset, we use 10% for validation, which involves optimizing hyperparameters related to the loss function and learning rate. For training, we use a batch size of 40, a learning rate of 0.01, a weight decay of 0.01, and an epoch count of 150. AdaGrad is used for the optimizer, and dropout is used for regularization. Additionally, flip augmentation is adopted to increase the amount of available training data.

Table 3.3: Pixel distribution for Indian Pines dataset with respect to each class.

Class No.	Class Name	Samples
1	Alfalfa	46
2	Corn-notill	1428
3	Corn-mintill	830
4	Corn	237
5	Grass-pasture	483
6	Grass-trees	730
7	Grass-pasture-mowed	28
8	Hay-windrowed	478
9	Oats	20
10	Soybean-notill	972
11	Soybean-mintill	2455
12	Soybean-clean	593
13	Wheat	205
14	Woods	1265
15	Buildings-Grass-Trees-Drives	386
16	Stone-Steel-Towers	93

3.5.1.1 Indian Pines

The Indian Pines dataset contains 145×145 pixels with 224 spectral bands. As suggested in [44], we discard the water absorption bands, which results in a modified dataset consisting of 200 bands. Each pixel in the dataset is categorized into 16 classes. The detailed distribution of pixels in the dataset can be found in Table 3.3. The Indian Pines dataset is illustrated as in Figure. 3.10 and 3.11

3.5.1.2 Pavia University and Center

The Pavia University and Pavia Center datasets are two datasets captured by flight over the town of Pavia in northern Italy. The Pavia University dataset has 103 spectral bands, while Pavia Center dataset has 102 spectral bands after 12 and 13 bands are removed due to noise, respectively [45]. The Pavia University dataset has a spatial di-

Figure 3.10: An illustration of the Indian Pines dataset in RGB representation.

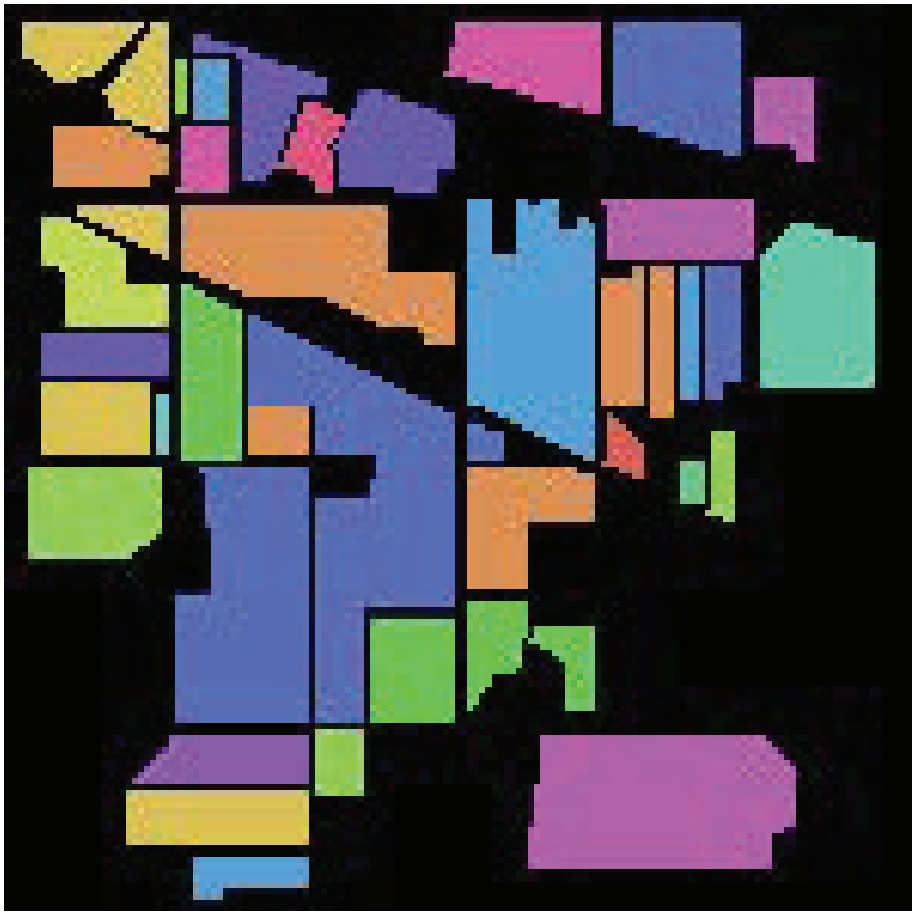


Figure 3.11: Ground truth information of the 16 classes in the Indian Pines dataset.

Table 3.4: Pixel distribution for Pavia University dataset with respect to each class.

Class No.	Class Name	Samples
1	Asphalt	6631
2	Meadows	18649
3	Gravel	2099
4	Trees	3064
5	Painted metal sheets	1345
6	Bare Soil	5029
7	Bitumen	1330
8	Self-Blocking Bricks	3682
9	Shadows	947

Table 3.5: Pixel distribution for Pavia Center dataset with respect to each class.

Class No.	Class Name	Samples
1	Water	824
2	Trees	820
3	Asphalt	816
4	Self-Blocking Bricks	808
5	Bitumen	808
6	Tiles	1260
7	Shadows	476
8	Meadows	824
9	Bare Soil	820

mension of 340 x 610 pixels. The Pavia Center dataset has a spatial dimension of 715 x 1096 pixels, and the center stripe of 381 pixels wide is removed from the original dataset, due to little to no information presented. Both datasets contain 9 classes of area. Detailed distributions of pixels in these two datasets can be found in Table 3.4 and Table 3.5. The Pavia University dataset is illustrated in Figure. 3.12 and Figure. 3.13, and the Pavia Center dataset is illustrated in Figure. 3.14 and Figure. 3.15.

3.5.1.3 Botswana

The Botswana hyperspectral dataset was captured by the NASA EO-1 satellite over the Okavango Delta, Botswana in 2001–2004. The original version of the dataset contains

Figure 3.12: An illustration of the Pavia University dataset in RGB representation.

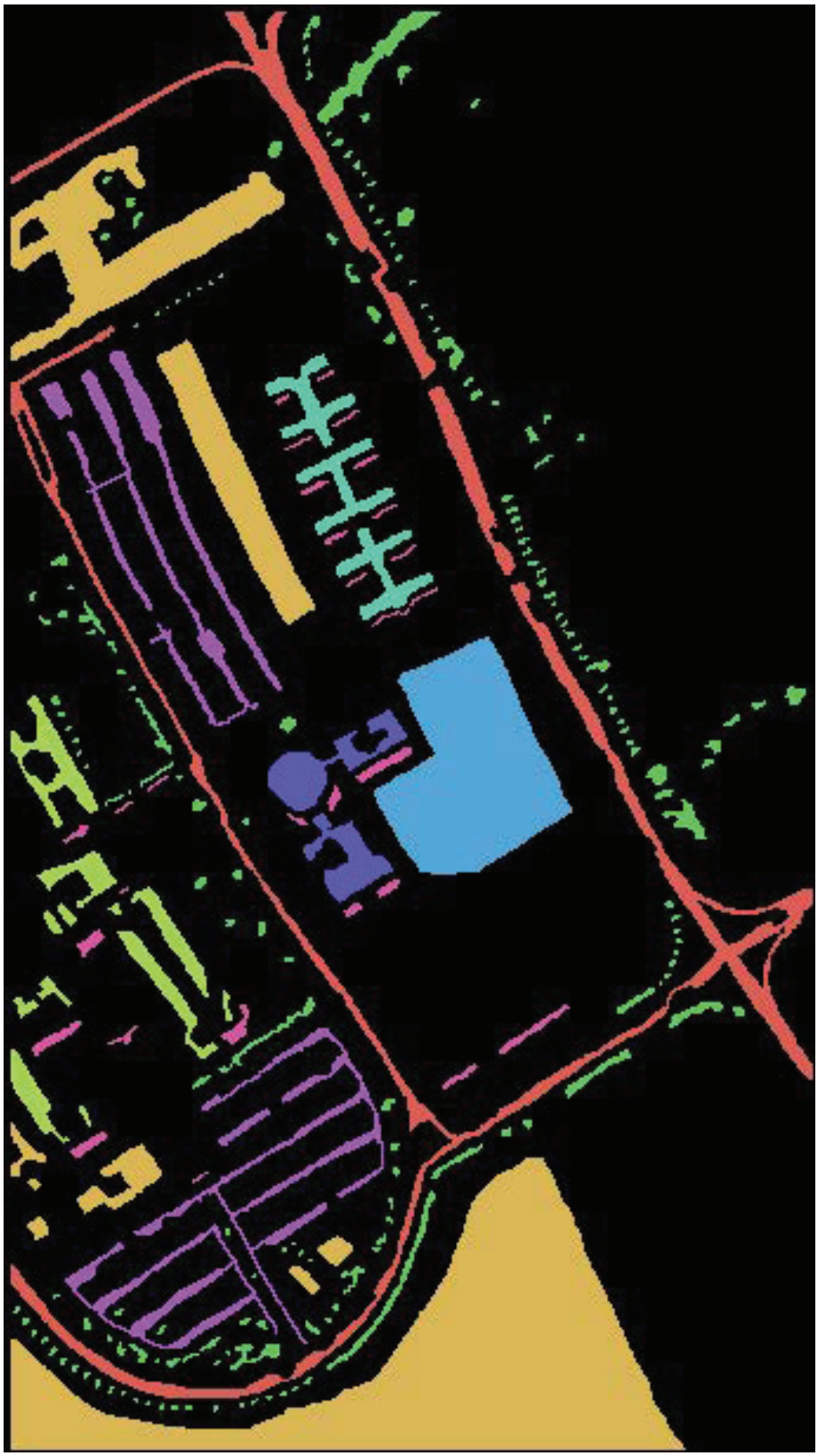


Figure 3.13: Ground truth information of the 9 classes in the Pavia University dataset.

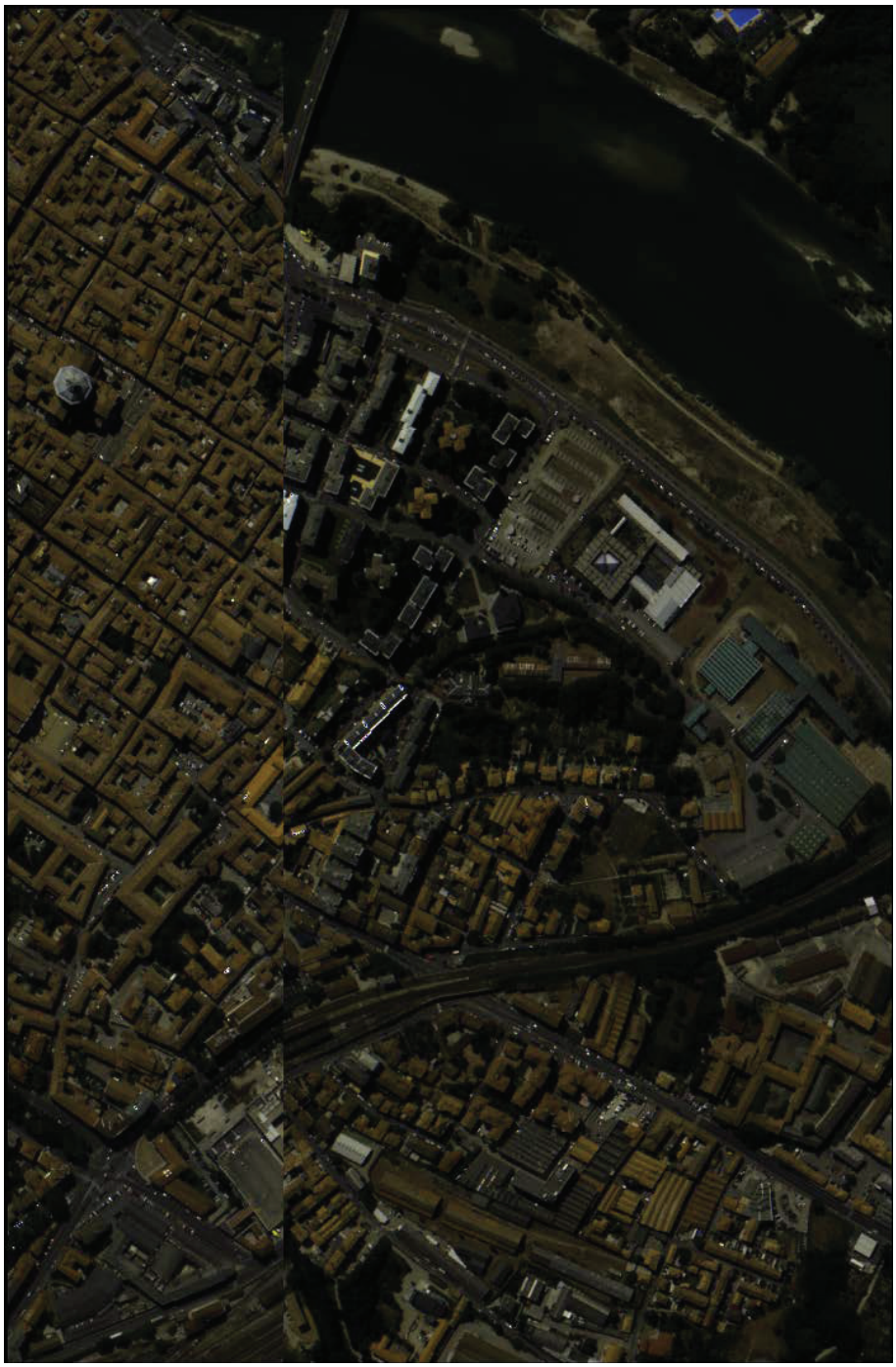


Figure 3.14: An illustration of the Pavia Center dataset in RGB representation.

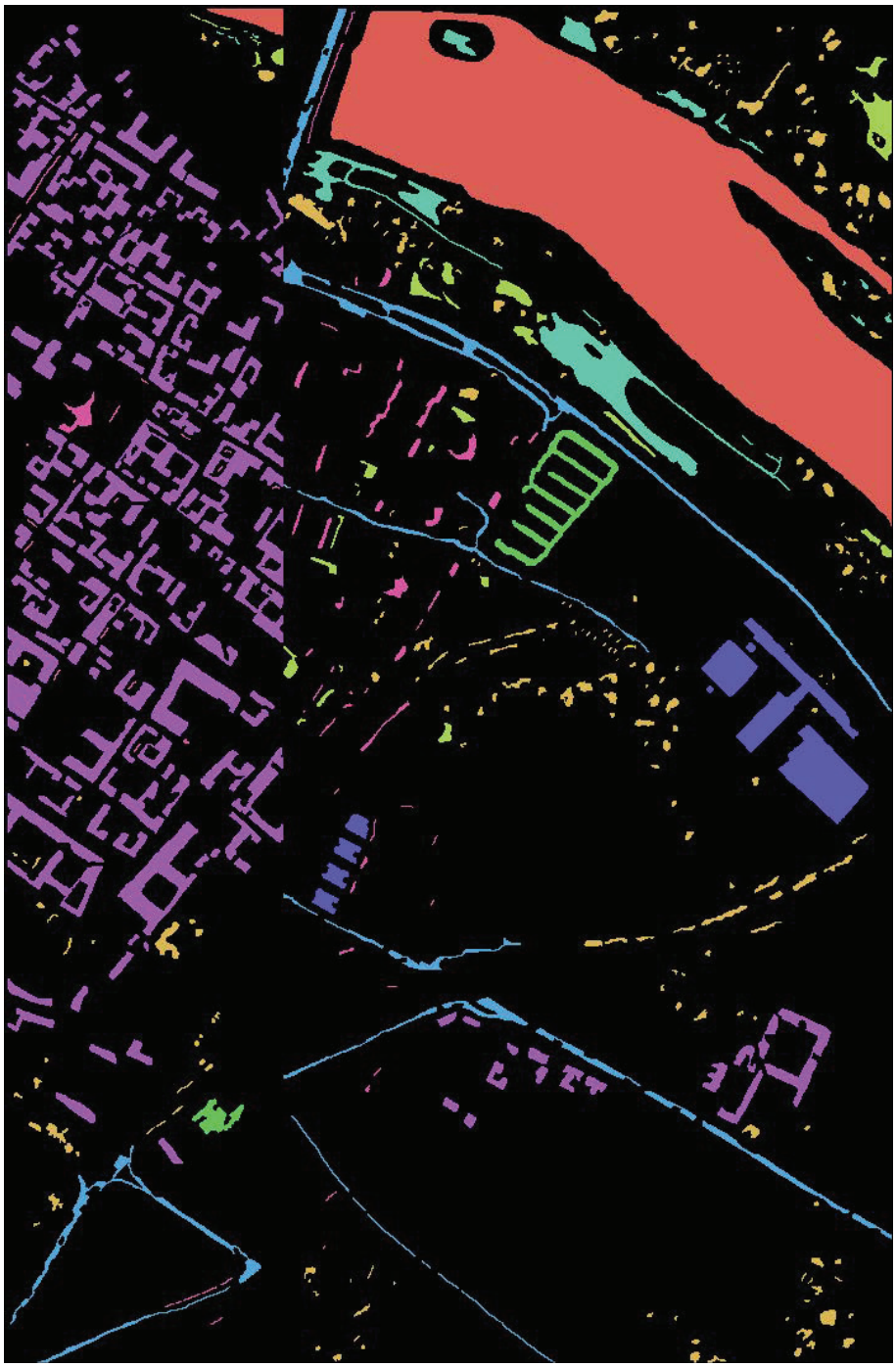


Figure 3.15: Ground truth information of the 9 classes in the Pavia Center dataset.

Table 3.6: Pixel distribution for Botswana dataset with respect to each class.

Class No.	Class Name	Samples
1	Water	270
2	Hippo Grass	101
3	Floodplain Grasses 1	251
4	Floodplain Grasses 2	215
5	Reeds 1	269
6	Riparian	269
7	Firescar 2	259
8	Island interior	203
9	Acacia woodlands	314
10	Acacia shrublands	248
11	Acacia grasslands	305
12	Short mopane	181
13	Mixed mopane	268
14	Exposed soils	95

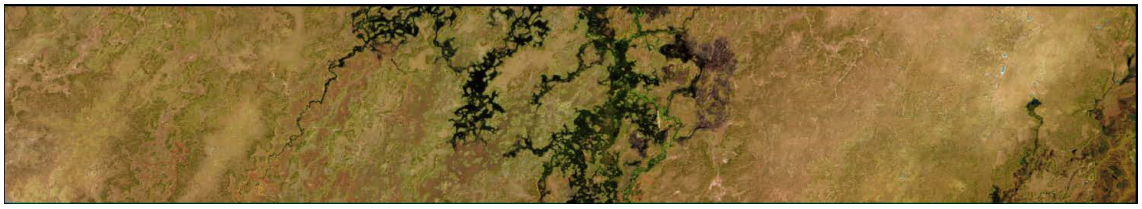


Figure 3.16: An illustration of the Botswana dataset in RGB representation.

1496×256 pixels with 242 spectral bands [46]. From the original version, 97 out of the 242 spectral bands are removed to eliminate effects of water absorption [47]. The dataset that we use in our experiments is the resulting dataset after removal of the 97 bands. The detailed distribution of pixels in the dataset can be found in Table 3.6. The Botswana dataset is illustrated in Figure. 3.16 and Figure. 3.17.

Figure 3.17: Ground truth information of the 14 classes in the Botswana dataset.

3.5.2 Experimental Setup

Except where specified otherwise, we repeat each experiment 50 times and report the average values across the 50 trials. As a resource-constrained platform on which to evaluate VBIC, we use the same Android mobile phone (OnePlus 7 Pro) from previous experiments. We use the PyTorch (Version 1.9) machine learning library for network implementation and training.

3.5.3 Results

Since the results for the Indian Pines and Pavia University datasets have been presented in Section 3.4, the rest of this section will focus on experimental results using the Pavia Center and Botswana datasets.

Table 3.7 presents the relationship between model size and classification accuracy. As depicted in the training scheme described in Algorithm 1, the model size increases with higher values of n_c, representing the number of usable spectral bands associated with the HSI dataset. The candidate values of n_c used in the experiments include 30, 60, 90, and *Maximum*, which represents the maximum number of usable spectral bands. Consistent with the results obtained from the Indian Pines and Pavia University datasets, increasing the value of n_c leads to larger model sizes and generally improved accuracy for VBIC, as demonstrated in Table 3.7 for the Pavia Center and Botswana datasets.

Moreover, the throughput, as shown in Table 3.8, Figure. 3.18, and Figure. 3.19, and the peak memory usage, as illustrated in Table 3.9, Figure. 3.20, and Figure. 3.21, follow similar trends as discussed in Section 3.4. These findings further support the conclusion

Table 3.7: Model size and accuracy on the Pavia Center and Botswana hyperspectral image datasets.

	Pavia Center		Botswana	
n_c	Model size	Accuracy	Model size	Accuracy
30	18,042	87.84%	24,447	88.92%
60	45,210	95.08%	64,415	96.54%
90	73,402	96.31%	105,407	97.79%
Maximum	89,306	98.18%	180,191	99.85%

Table 3.8: Processing throughput on the Pavia Center and Botswana hyperspectral image datasets.

	Pavia Center		Botswana	
n_c	Average Throughput (PC/s)	Standard Deviation	Average Throughput (PC/s)	Standard Deviation
30	52,328.08	518.27	51,093.28	435.06
60	16,205.28	83.62	16,381.68	152.7
90	9,611.84	67.33	9,643.2	63.51
Maximum	5,327.28	69.85	4,100.32	32.21

that VBIC exhibits runtime adaptiveness under different numbers of selected bands n_c, allowing for a systematic trade-off between runtime performance metrics and accuracy.

Table 3.9: Peak memory consumption on the Pavia Center and Botswana hyperspectral image datasets.

	Pavia Center	Botswana
n_c	Peak memory usage (MB)	Peak memory usage (MB)
30	459	340
60	479	394
90	593	423
Maximum	609	470

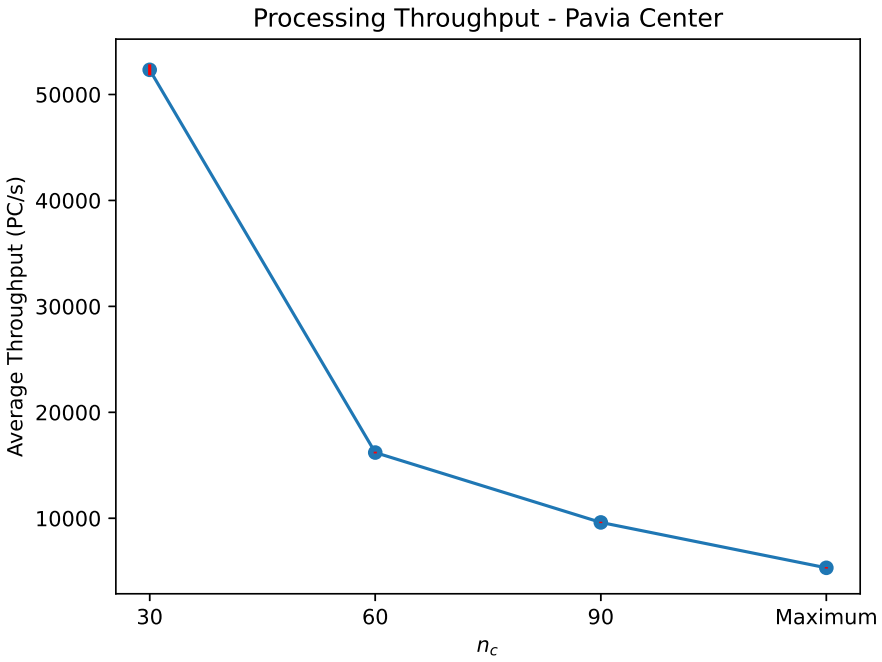


Figure 3.18: Throughput comparison for VBIC under different values of n_c with the Pavia Center dataset.

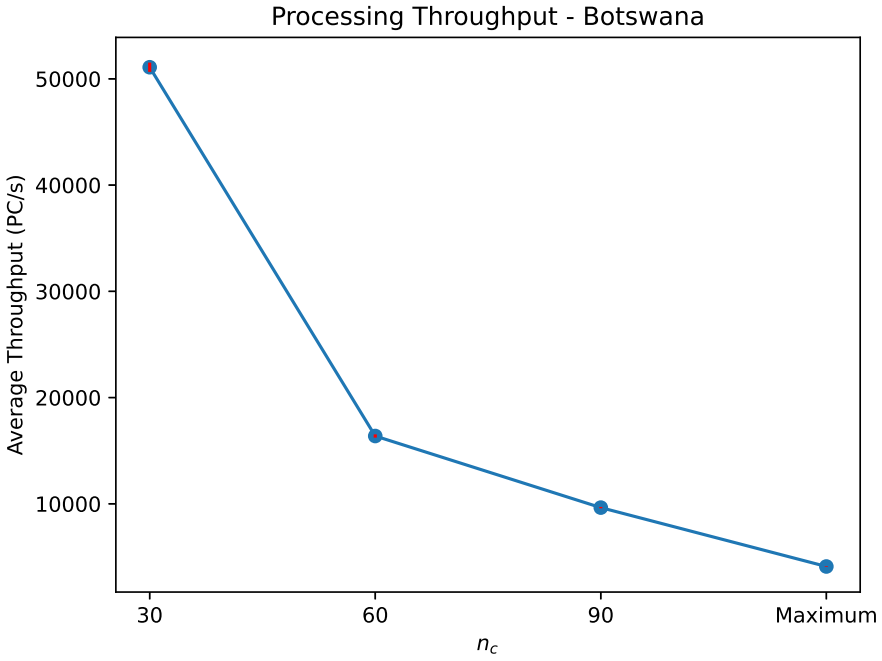


Figure 3.19: Throughput comparison for VBIC under different values of n_c with the Botswana dataset.

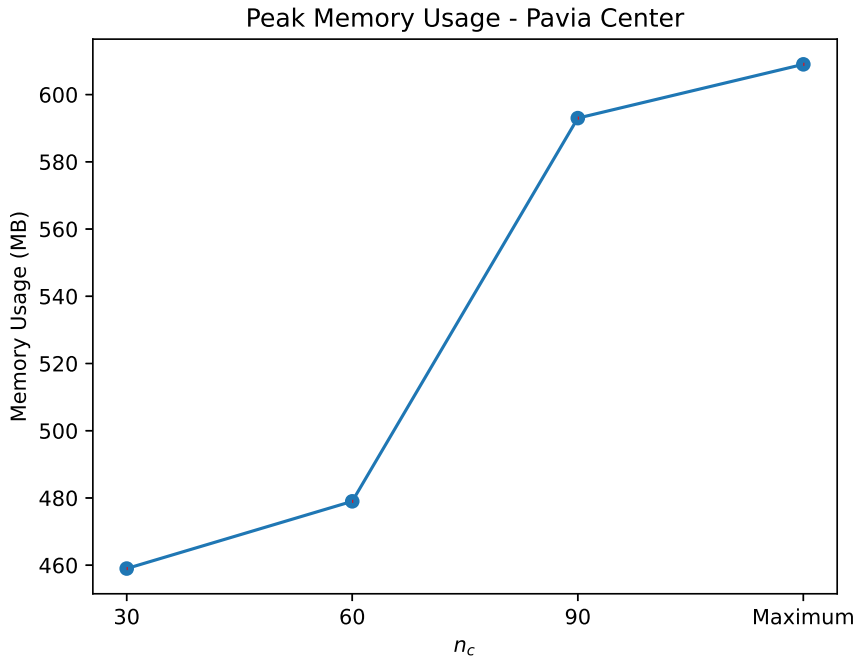


Figure 3.20: Peak memory usage comparison for VBIC under different values of n_c with the Pavia Center dataset.

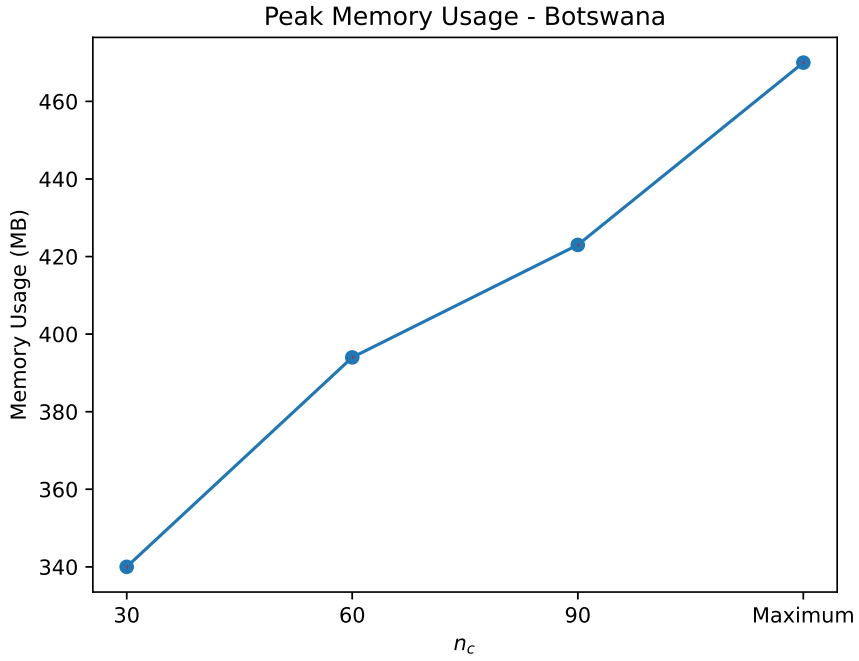


Figure 3.21: Peak memory usage comparison for VBIC under different values of n_c with the Botswana dataset.

3.6 Summary

In this chapter, we have developed a framework for hyperspectral image classification called Variable Band Image Classification (VBIC). VBIC integrates adaptive DNN-based image analysis with real-time, resource-constrained processing. The framework provides novel DDDAS capabilities by enabling dynamic self-adaptation of the deployed DNN configuration to maximize hyperspectral image analysis accuracy subject to operational requirements that may vary dynamically. We have demonstrated the utility of VBIC through an implementation on an Android platform, and experiments using relevant datasets. Useful directions for future work include exploration of VBIC implementations on embedded GPUs and neural network accelerators, and investigation of systematic methods for selecting the set of options for band subset sizes.

Chapter 4

LCIP: A Retargetable Framework for Optimized CNN Inference

In Chapter 3 we presented VBIC, a novel implementation designed for hyperspectral image (HSI) classification tasks. In this chapter, our focus shifts to the design of a software tool that specifically addresses four general issues related to Convolutional Neural Network (CNN) based inference on resource-constrained platforms: retargetability, resourceability, efficiency, and configurability. In this chapter, we present a software package, named LCIP, that addresses these four aspects of CNN implementation in an integrated manner. LCIP facilitates enhanced design processes for deploying CNNs on resource-constrained platforms.

Material in this chapter was published in partial, preliminary form in [48].

4.1 Introduction

Deep neural networks (DNNs) are ubiquitous in deployment of embedded signal and information processing systems for many challenging applications. Deep configurations of convolutional neural networks (CNNs) in particular have been used very widely in recent years and continue to be of increasing interest. However, along with advances in DNN methods comes an interest in applying DNNs to increasingly complex problems that are deployed at the network edge, where availability of computational resources and energy budgets are limited. Motivated by these trends, we develop in this chapter a novel

software tool, called the Lightweight-dataflow-based CNN Inference Package (LCIP), for streamlining deep CNN implementation. LCIP is designed to address the challenges of deploying highly complex CNN inference systems on resource constrained platforms. While we demonstrate LCIP using networks for image classification, LCIP provides a general framework that can be applied to other types of CNN applications as well.

Because of the high computational complexity introduced by the convolution operation, the key operation in CNNs, developing and deploying state-of-the-art CNN models at the network edge is challenging. Moreover, the design spaces involved in edge-based CNN deployment are very complex, and call for systematic approaches to enable rapid evaluation of alternative design configurations. These design spaces include diverse factors such as network architecture (topology) selection, processing device selection, and scheduling configurations for orchestrating network execution on parallel hardware. A major objective of LCIP is to facilitate design space exploration of resource-constrained CNN implementation through systematic methods that are based on dataflow techniques.

By enabling more efficient design space exploration and operating at a high level of abstraction, LCIP contributes to improving the system design workflow, and allows designers to dedicate more focus towards the development of use cases and applications without being distracted by low-level particularities (e.g., different application programming interfaces, build tools, etc.) imposed by the machine learning (ML) frameworks that the CNNs are designed and trained with. Related work on improving inference implementation for DNNs includes Apache TVM [49], a DNN inference engine that applies platform-specific optimization to pre-trained networks, as well as the approach for CNN inference on embedded CPUs-GPU multiprocessor systems-on-chip (MPSoCs) proposed

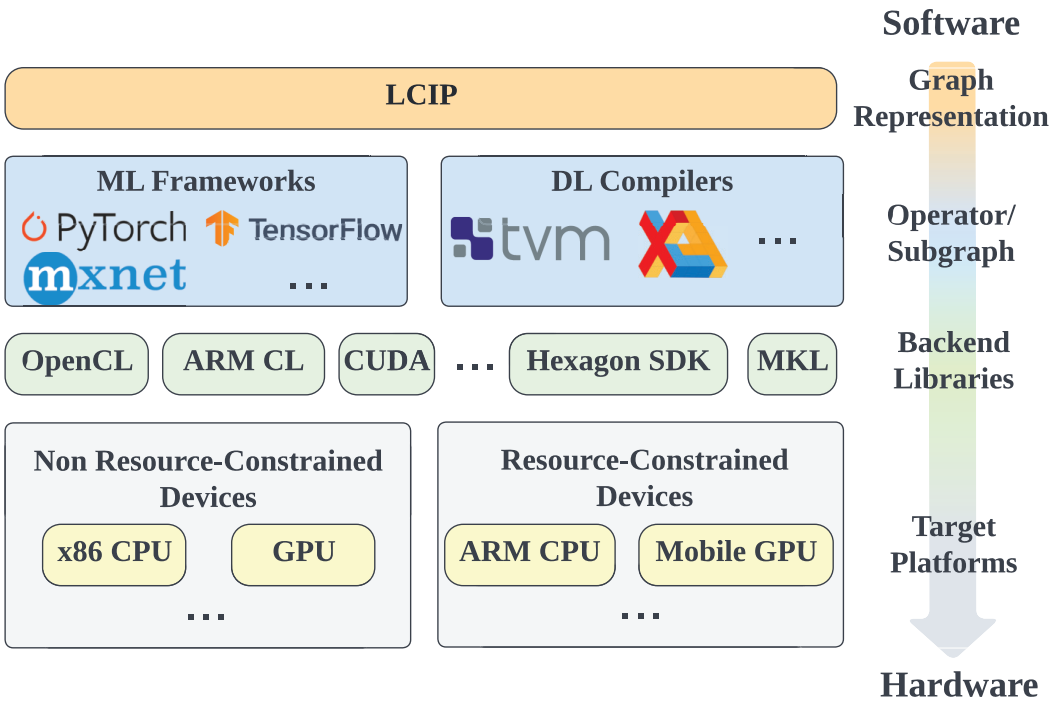


Figure 4.1: Overview of LCIP.

in [50].

In this chapter, we present a novel software tool, called the Lightweight-dataflow-based CNN Inference Package (LCIP), which addresses the following issues of edge-based CNN implementation in an integrated manner: retargetability, resourceability, efficiency, and configurability. Figure. 4.1 provides a hierarchical overview of LCIP, and how it interacts with libraries, hardware, and other machine learning frameworks.

Here, by *retargetability*, we mean the ability to efficiently implement designs on different types of processing platforms. Retargetability in LCIP is achieved by its application of the Lightweight Dataflow Environment (LIDE) [41], which is a software package for dataflow-based design and implementation of embedded signal and information processing systems. LIDE is based on a compact application programming interface (API) that

is defined in terms of abstract dataflow principles, and can be implemented concretely in arbitrary platform-oriented programming languages (e.g., C, C++, CUDA, OpenCL, and Verilog), and mapped to different hardware targets, such as x86 CPUs, ARM CPUs, CUDA-enabled NVIDIA GPUs, and field programmable gate arrays.

For example, LIDE has been used as a foundation for software synthesis of dataflow graphs onto CPU-GPU platforms [51]. Through the dataflow based implementation features inherited from LIDE, LCIP allows DNNs to be modeled precisely in terms of dataflow principles, and mapped into implementations on diverse processing platforms.

Resourceability is a term that we introduce to describe the flexibility to integrate with different frameworks for training ML models. In the design flows enhanced by LCIP, such frameworks provide the "source" or input models for the implementation process. Different design teams have different preferences in terms of ML frameworks (e.g., TensorFlow [52] versus PyTorch [53]) or deep learning compilers (e.g., Apache TVM [49]) so flexibility in working with pre-trained CNNs obtained from a variety of commonly-used ML frameworks is useful in enhancing the utility of LCIP.

Efficiency of CNN inference is crucial in edge implementation. Inefficient implementations on resource-constrained platforms may lead to unacceptable inference delays or excessive energy consumption. In our experiments, we demonstrate efficiency improvements provided by LCIP in terms of inference throughput, which is an important metric for many types of edge-targeted neural networks.

Configurability is perhaps the most distinguishing aspect of LCIP. In this chapter, we mean a specific type of configurability in which the mapping of the high-level dataflow of a DNN can be flexibly scheduled across available processing resources. This schedul-

ing process includes partitioning the DNN layers across different hardware resources or threads and controlling the exploitation of task-level and pipeline parallelism across DNN layers. Configurability is crucial to design space exploration since the number of alternative scheduling configurations for a dataflow graph in general grows rapidly (super-polynomially) with the size of the graph, and the dataflow schedule has a large impact on metrics that include throughput, latency, memory requirements, and energy consumption (e.g., see [21]).

LCIP provides an integrated approach to addressing resourceability, efficiency, retargetability, and configurability in edge-based CNN inference. LCIP offers a bridge between dataflow modeling and existing ML inference frameworks by providing flexible high-level graph scheduling for CNN inference on resource-constrained devices. The high level dataflow methods applied by LCIP can be adapted systematically to different target platforms and platform-specific libraries based on designer specifications and available hardware resources. Through an extensive experimental evaluation, we compare LCIP to commonly used ML frameworks with respect to key performance metrics, and we demonstrate significant performance improvements delivered by LCIP compared to existing state-of-the-art methods.

The general design flow of deploying a model in LCIP starts with an *importer*, which parses the pre-trained network and generates a LIDE-compatible graph representation of the original CNN. The generated LIDE graph can then be further profiled and optimized with various types of parallelism, such as pipeline parallelism, task parallelism, intra-operator parallelism, and data parallelism. Due to the design flexibility introduced by dataflow modeling in LCIP, arbitrary combinations of these types of parallelism can be

easily explored as options for performance optimization. Further performance enhancement can be explored in conjunction with the aforementioned optimizations, including but not limited to platform-specific optimizations, distributed inference, and heterogeneous inference.

LCIP is capable of providing CNN inference on multiple devices connected over a network, with the inference load distributed across the available devices. The load on each device is balanced to optimize overall inference throughput. This feature is crucial to deploying CNNs efficiently on resource-constrained devices because a single device might not be capable of achieving reasonable runtime performance, due to the more energy-efficient and less computationally-powerful design of resource-constrained devices. However, an ensemble of such devices can collectively contribute to the target application with the CNN inference task partitioned across the different devices in the ensemble.

4.2 Related Work

Many attempts have been made to improve the feasibility and performance of deploying CNN-based systems across different hardware platforms. For example, Apache TVM [49] aims to create an inference compiler engine for pre-trained DNNs from popular ML frameworks for efficient deployment. TVM also targets different hardware platforms, including CPUs, GPUs, and accelerators, by utilizing the unique computing kernels available for the platforms. TVM operates by converting pre-trained DNNs that are input to the tool into TVM's proprietary intermediate representation, and then performing platform-specific optimizations on the intermediate representation.

Similar to LCIP, TVM also models DNNs as dataflow graphs; what makes LCIP unique in this respect is that LCIP allows dataflow graph schedules to be designed independently from the dataflow actors and edges, which enables flexible network-level scheduling. For example LCIP allows great freedom to the designer in partitioning DNN layers across available processors or threads, while such flexibility is not available in TVM. This form of design flexibility provided by LCIP is what we have referred to as *configurability* in Section 4.1. With this form of configurability, LCIP can be viewed as complementary to TVM. Whereas TVM's emphasis is on mapping CNN layers efficiently into optimized platform-specific implementations, LCIP focuses on providing configurability for network-level (inter-layer) design optimization and trade-off exploration.

Edge processing devices and other types of resource-constrained processors are becoming increasingly popular due to advancements in CPU and GPU technology for computation- and energy-constrained applications. As a result, these devices are more than capable of deploying applications that are equipped with neural networks.

Many tools provide capabilities for optimizing neural network inference on edge devices. For example, TensorFlow Lite [52] is a mobile library that is specifically designed for deploying neural networks on edge devices with support for post-training model optimization at the operator level. However, TensorFlow Lite does not support ML frameworks other than TensorFlow directly, making it difficult for users that are unfamiliar with TensorFlow to utilize the full capability of TensorFlow Lite. PyTorch Mobile [53] is another notable example of an edge inference library that addresses the model deployment problem for mobile devices. PyTorch Mobile utilizes various low-level compute libraries, such as QNNPACK [54], to streamline model deployment on mobile devices.

Similar to TensorFlow Lite, PyTorch Mobile focuses on optimizing pre-trained models at the operator level, and also supports models that are trained only with the parent framework (PyTorch in this case). LCIP, on the other hand, supports different ML frameworks and neural network compilers to help address problems associated with model compatibility and provides designers with greater flexibility in their choice of ML frameworks. Additionally, LCIP provides sophisticated graph-level optimizations to pre-trained model deployment, which provides more flexibility for design optimization. The graph-level optimizations provided by LCIP utilize different levels of parallelism; the optimizations are enabled by the dataflow modeling methods employed in LCIP, which are discussed in detail in Section 4.3.

In scenarios where CNNs are deployed on edge devices, there usually are unique advantages of choosing edge devices over more powerful platforms; such advantages include factors such as mobility and connectivity. However, the limited computing capability and power budget in edge devices often result in undesirable runtime performance for many applications, especially for real-time applications. To alleviate these restrictions, offloading inference tasks from the targeted edge device to other network devices is a potential solution. In such a distributed deployment, each individual device is only responsible for partial CNN inference, which has the potential to result in significantly higher inference throughput compared to what can be provided by a single edge device operating in isolation.

Hu et al. [55] propose a novel distributed inference mechanism called *EdgeFlow*, which utilizes a progressive model-partitioning algorithm to partition a DNN model by layers and balance the computational load on across a set of cooperating edge devices. Li

et al. [56] propose a distributed inference framework with a pool of GPUs having different specifications, and compare the inference latency against a homogeneous pool of GPUs. By building on the use of dataflow modeling and LIDE, distributed and heterogeneous inference in LCIP can be achieved in various ways, such as by applying graph-level data parallelism, where each device contains a complete copy of the given CNN graph and is responsible for processing different inputs at any given times, or by applying pipeline parallelism, where each device is only responsible for part of the CNN graph.

The application of dataflow techniques to neural network design and implementation has been researched actively in recent years. For example, Xie et al. introduced methods for modeling DNNs in terms of synchronous dataflow (SDF) graphs, and applying these models to derive efficient multicore CPU implementations [57]. SDF is widely-used in model-based signal processing system design due to its support for compile-time design optimization and verification [58, 21]. Minakova et al. introduced a methodology that provides dataflow modeling of CNNs to facilitate exploitation of both task-level and data-level parallelism [50]. Their work utilized the flexibility of state-of-the-art GPU-enabled embedded systems, combined with the dataflow modeling of CNNs, to achieve high-throughput CNN inference on low-power devices. Luenemann et al. presented methods for translating from neural network classifiers to SDF representations, and applying the resulting SDF representations for design space exploration and optimization on embedded low power systems [59]. Sankaranarayanan et al. proposed a dataflow- and distributed-DNN-based Internet of Things (IoT) and edge computing model that offers low latency and improved accuracy in data generation [60]. Venieris et al. presented an end-to-end framework for the optimized mapping of CNNs onto field-programmable

Table 4.1: Comparison of LCIP along with several previously-developed neural network inference systems.

Reference Works	Dataflow Semantics	Resourceability	Retargetability	Configurability
TVM [49]	✗	✓	✓	✗
Minakova et al. [50]	✓	✗	Limited	✓
Luenemann et al. [59]	✓	✗	✗	✗
Veeramanikandan et al. [60]	✗	Unknown	✓	✓
Venieris et al. [61]	✓	✓(Only for FPGAs)	✓(Only for FPGAs)	✓
Song et al. [62]	✓	✓	✗	✗
Xie et al. [57]	✓	✓	✓	✗
LCIP (Proposed)	✓	✓	✓	✓

gate arrays [61]. The mapping process utilizes SDF modeling along with multiobjective optimization techniques to derive efficient hardware realizations. Song et al. proposed a dataflow-based synthesis tool called DFSynthesizer, which is designed for deploying Spiking Neural Networks (SNNs) on low power neuromorphic hardware platforms [62].

Table 4.1 provides a comparison of LCIP along with several previously-developed neural network inference systems. The comparison is presented in terms of four dimensions — Dataflow Semantics, Retargetability, Resourceability, and Configurability. Efficiency is a major consideration in all of the tools, so it is not represented explicitly as a separate column in the table. The use of dataflow semantics (Column 2) is useful for important reasons in addition to the connection to configurability (see Section 4.1). For example, through fundamental properties inherited from Kahn process networks, important classes of dataflow representations (including the SDF representations that we apply in LCIP) provide guarantees on key implementation properties, such as determinate execution, deadlock avoidance, and bounded memory operation [20, 63].

✓ and ✗ in the entries of Table 4.1 represent "Yes" and "No" correspondingly. An entry of "Unknown" in the table indicates that it is not clear from the associated reference whether the tool supports the associated design dimension.

From Table 4.1, we see that LCIP is the only tool among those listed that simultaneously provides dataflow semantics, resourceability, retargetability, and schedule configurability. Additionally, we would like to emphasize that while configurability is supported by some of the other referenced systems, LCIP is unique in its support for the exploration of multi-dimensional design spaces. In contrast, existing systems primarily target system throughput or latency as the key performance metric. The capability of LCIP to assist in exploring trade-offs between multiple metrics is demonstrated concretely in Section 4.4.

4.3 Methods

In this section, we provide details of LCIP, including the overall architecture, dataflow modeling, implementation details, and graph-level optimization.

4.3.1 Architecture

In the design flow associated with LCIP, a pre-trained CNN is modeled as an SDF graph, where each layer or block of layers in the network corresponds to a vertex in the graph, and connections between layers or blocks are modeled as edges. In SDF modeling of signal and information processing systems, vertices are called *actors*, and edges correspond to First-In-First-Out (FIFO) communication channels [58]. Data values that pass through the edges are abstracted as *tokens*. In many kinds of dataflow analysis, we are interested only in the numbers of tokens involved rather than the specific values of the tokens. For detailed background on dataflow modeling in this context, we refer readers to [21].

The designer can configure the grouping of selected subsets of adjacent CNN layers into clusters of actors as desired. In LCIP, such clustering of actors constrains the mapping and scheduling of the actors onto processing units — in particular, all actors in a given cluster are mapped to the same CPU thread or accelerator. Thus, through the process of clustering, designers or higher level design tools can influence and explore key implementation details associated with heterogeneous multicore platforms.

LCIP is designed to work with two major types of companion tools: (1) ML frameworks that support design of pre-trained CNNs, and (2) low-level compute libraries (LLCLs) that are utilized by ML frameworks to accelerate network inference components on targeted devices. From the user's point of view, once the desired ML framework is installed with compatible LLCLs, it is only necessary to interact with the LCIP version of the pre-trained CNNs — and its underlying SDF representation — without having to concern oneself about how LCIP interacts with the ML framework or the LLCLs. The layer of abstraction provided by LCIP ensures that the user is only responsible for network modeling and schedule configuration in LCIP and relieves users from tedious low-level details that are involved in deploying CNNs on different hardware platforms.

Figure. 4.2 illustrates the system architecture of LCIP, as well as the details of each component within LCIP. The design flow of LCIP starts with an importer that converts a pre-trained CNN model from a supported ML framework to an LCIP dataflow graph. The importer accepts two inputs to create the dataflow graph: the *pre-trained CNN model* and *LCIP mapping*. The pre-trained CNN model can be obtained from a supported ML framework with appropriate preprocessing steps, which are easily automated in LCIP, to convert the model into a form that is suitable for LCIP. For example, to import a PyTorch

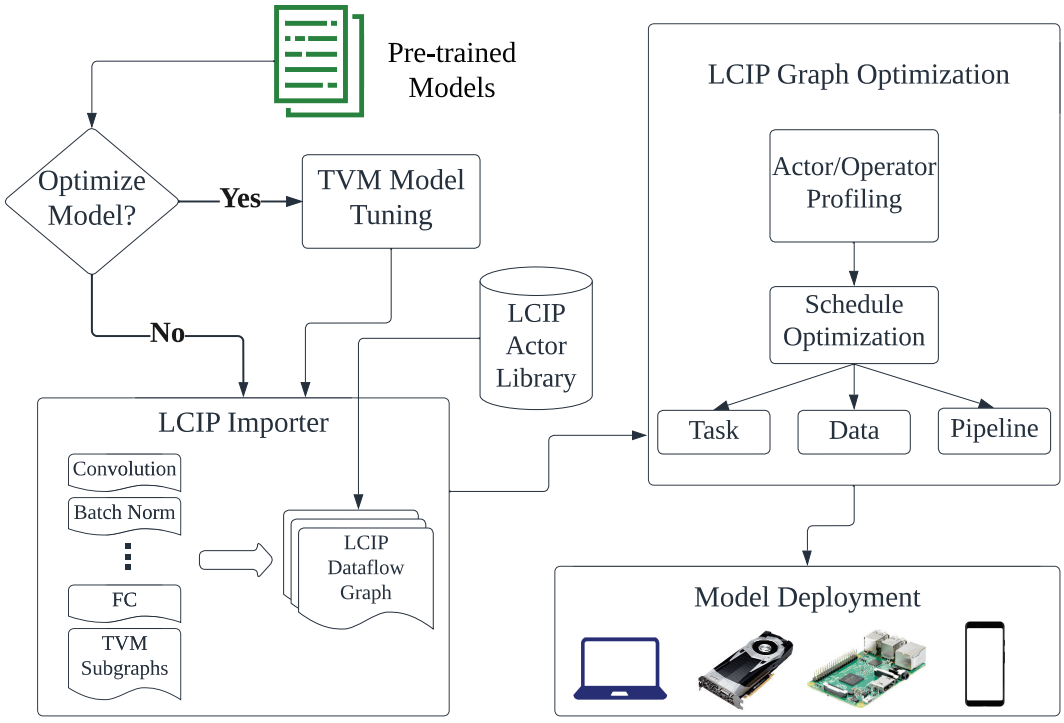


Figure 4.2: A system diagram that illustrates key components of LCIP, and the workflow involved in deploying a CNN with LCIP.

model into LCIP, the LCIP importer needs access to a scripted or traced pre-trained model (a standalone model file that can be used for deployment independently) that is obtained from PyTorch.

The LCIP mapping is a one-to-one mapping between the operators of the pre-trained CNN and LCIP dataflow actors. Unlike other tools/frameworks that involve a proprietary version of each operator for the conversion, LCIP is agnostic to any particular set of operator implementations. The particular LLCL that the user applies with LCIP provides the operator implementations, while LCIP provides the dataflow abstractions and dataflow-based manipulations for integrating the operators. Thus, an LCIP actor can be viewed as a dataflow-based wrapper around an operator implementation from the given LLCL. As its output, the LCIP importer generates a dedicated dataflow graph in C++ based on the dataflow modeling features in LIDE.

Once the LCIP dataflow graph is constructed, the user can focus on designing a dataflow schedule for the given graph or experimenting with alternative schedules. The schedule depends on the topology of the graph as it specifies the order in which the graph is traversed at runtime, but through relevant properties of dataflow modeling (e.g., see [21]), the scheduler implementation is cleanly separated from actor implementation in LCIP. In particular, as long as the interface behavior of the actors in unchanged, changes to a schedule require no changes to actor implementation, and similarly, actor implementations can be fine-tuned without requiring any changes to a given schedule.

It is during this step of the design process that LCIP optimizes the inference performance of the provided CNN. The optimization is facilitated by the semi-automated processes of dataflow schedule implementation, and mapping of actors to different pro-

cessors or threads, along with the flexibility to integrate with a variety of LLCLs. More details on the processes of schedule design and graph-level optimization are discussed in Section 4.3.4.

4.3.2 Dataflow Modeling

Figure. 4.3 illustrates an LCIP model of the VGG11 network [64], which we use in the experiments that we present in this chapter (Section 4.4). In the figure, each white rectangle corresponds to a distinct actor in the dataflow graph; the name of the operator associated with each actor is enclosed within the rectangle. The Conv and BN modules are actors that perform 2D convolution followed by batch normalization, respectively. The ReLu actor performs a Rectified Linear Unit operation. The MaxP actor shown is a max pooling operator, and similarly, the AvgP actor performs average pooling. The Linear actor represents a linear or fully connected layer, and the Class actor encapsulates the classifier that provides the final classification outcome for a given input image to the network. Figure. 4.3 shows a version of VGG-11 where the graph has been partitioned into three segments for placement onto three distinct threads of a targeted multicore processor. More details related to experimentation with partitioning are discussed in Section 4.4.

The data type (token type) associated with each edge in Figure. 4.3 is *pointer to tensor*, where `tensor` is a suitably defined type for multi-dimensional data in LCIP. That is, pointers to the tensors flow through the dataflow graph rather than the tensors themselves. The dimensions of the tensors vary across the different edges in the dataflow graph but in all cases, the token size (in bytes) is simply the size of a pointer type. The dataflow

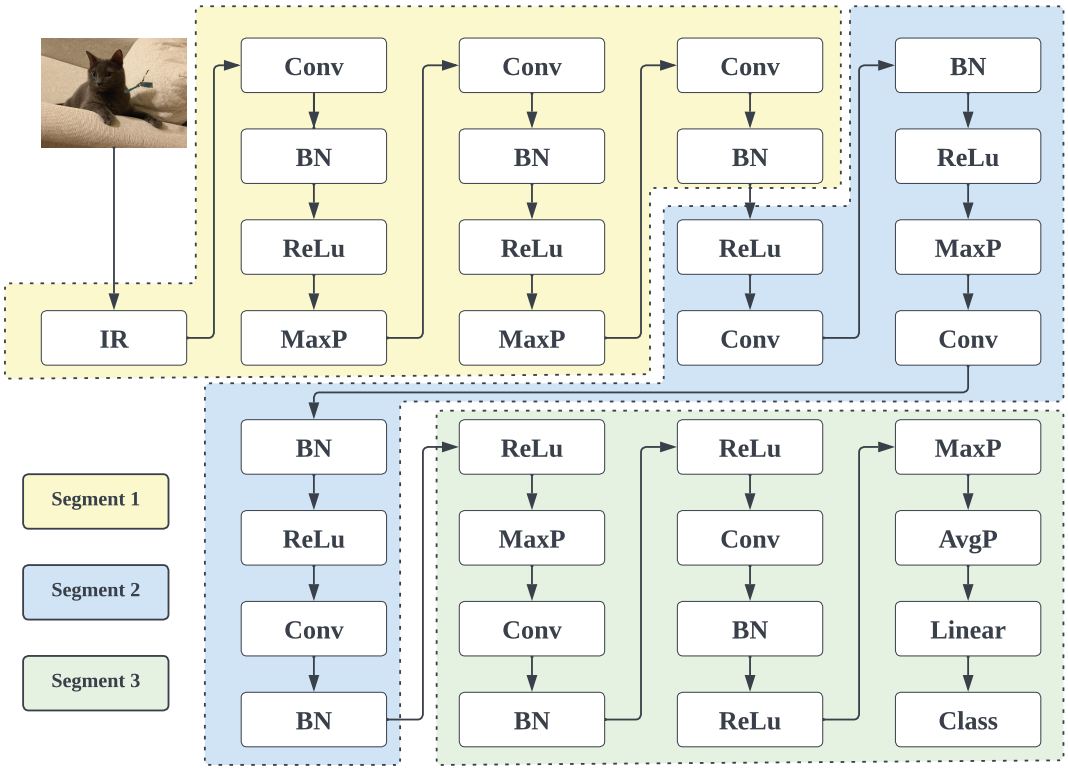


Figure 4.3: An example of the VGG-11 network modeled as a dataflow graph in LCIP.

graph is a *homogeneous* SDF graph, which means that on each execution (firing), each actor produces a single token on each output port and consumes a single token from each input port (in general SDF graphs, these production and consumption "rates" need not be restricted to just one token per firing).

4.3.3 Implementation

Dataflow-based neural network models in LCIP are implemented using LIDE [41], as described in Section 4.1. Actor design in LIDE is based on a form of dataflow called Core Functional Data Flow (CFDF). In CFDF, an actor is specified in terms of one or more operational modes, where the production and consumption rates for the actor are

constant in a given mode, but rates can vary across modes, thereby allowing for dynamic dataflow behavior. SDF is an easily-identifiable special case of CFDF where all modes of a given actor have identical production and consumption rates.

LCIP actors are initialized as part of the graph construction process for a CNN model. The construction process is carried out as part of the constructor of the C++ dataflow graph class in LCIP, and this constructor in turn calls the constructors for all of the actors in the graph to instantiate and initialize the actors. Actor initialization generally includes loading of any relevant parameters, such as pre-trained weights, from configuration files that are associated with the actor.

When an LCIP actor fires, a token that encapsulates a pointer to an input tensor is consumed from each input edge. The input tensors referenced in this way are used as operands to the ML-framework-specific operators, along with any actor parameters that are derived by the ML framework and loaded into the actor's state at actor initialization time. At the end of the firing, a pointer to each output tensor will be produced on the associated output edge. Most actors have only one output edge, so only one tensor pointer is produced in the typical case. Operator implementations used in LCIP actors are LLCL-specific and wrapped within the functions that execute the actor firings. The operator implementations are independent from LCIP and can be replaced by similar operators for alternative platforms/LLCLs as long as they are wrapped within the firing of an LCIP actor to properly interface the operator to the dataflow graph. This methodology allows for systematic integration with arbitrary LLCLs while adhering to a common, higher-level dataflow model abstraction so that dataflow-based design and analysis techniques can be reused and appropriately adapted across different ML-framework / LLCL combinations.

Listing 4.1: PyTorch-based implementation of the Conv2d actor in LCIP.

```
/* read input from  input fifo tensor_in */
torch::Tensor *input = nullptr;
lide_c_fifo_read(tensor_in, &input);

/* Conv2d operator from Torch */
out = torch::conv2d(*input,
                    *conv2d_weight,
                    *conv2d_bias,
                    stride,
                    padding);

/* write result pointer to output fifo
   tensor_out */
torch::Tensor *output = &(out);
lide_c_fifo_write(tensor_out, &output);
```

Torch-based LCIP Actor For example, suppose the VGG-11 network is trained with PyTorch. Then the PyTorch-specific operator for the Conv2d operator can be implemented within an LCIP actor as shown in Listing 4.1. Here, `conv2d` is the LLCL-specific operator from the PyTorch C++ API, as the namespace `torch` suggests. The exact operator used in an LCIP actor can be replaced by an operator from another ML framework, as long as a C/C++ API is provided such that the operator can be embedded within a C/C++ actor implementation in LCIP. Although the underlying LIDE framework supports several other implementation languages apart from C and C++, LCIP presently assumes a C/C++ implementation format. Extension of LCIP beyond C/C++ (e.g., to a hardware description language such as Verilog) is a useful direction for extending the tool in future work.

TVM-based LCIP Actor Unlike the Torch version of the LCIP actor described above, where an actor encapsulates a single CNN operator, the TVM-compatible actor encap-

sulates a subgraph of the original CNN graph. The reason for the difference between TVM and Torch versions of LCIP is that the two frameworks are oriented towards different objectives and use cases. When TVM is applied, the pre-trained CNN is already optimized at the intra-operator level for the target platform, as TVM utilizes available LLCLs on the target device to optimize the compute kernel of individual operators. Additionally, TVM incorporates some inter-operator optimizations, such as operator fusion. Therefore, from LCIP's perspective, providing more efficient graph-level optimization for TVM-compatible dataflow graphs becomes a highly complementary objective.

An example of an LCIP actor that implements a TVM subgraph can be found in Listing 4.2. When integrating TVM with LCIP, it is useful to combine subgraphs of operators into individual actors in cases where TVM applies optimizations across multiple operators in the subgraphs. Such a subgraph-encapsulated LCIP actor can be deployed on a wide range of target devices with support from TVM's backend compatibility, making it well-suited for CNN inference deployment on edge devices.

4.3.4 Graph-level Optimization

The efficiency and configurability aspects of LCIP follow naturally from the strict adherence to dataflow modeling principles, as enforced by use of LIDE as the underlying programming model. Modeling the CNN as a dataflow graph enables high-level schedule design, such as scheduling techniques that exploit strategic partitioning of the graph across different physical processing resources or threads and that exploit task-level parallelism and graph-level pipeline parallelism in different ways. Such an emphasis

Listing 4.2: TVM-based implementation of the conv2d actor in LCIP.

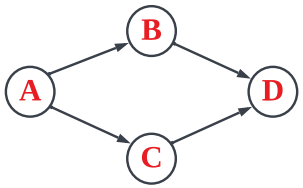
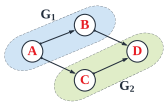
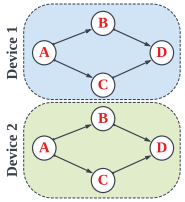
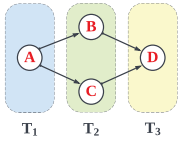
```
/* read input from input fifo tensor_in */
tvm::runtime::NDArray *input = nullptr;
lide_c_fifo_read(tensor_in, &input);

/* create inputs/outputs for specific targets */
TVMArrayAlloc(in_shape, in_ndim,
              in_dtype_code, in_dtype_bits,
              dtype_lanes, device_type,
              device_id, &inputNdarray);
TVMArrayAlloc(out_shape, out_ndim,
              out_dtype_code, out_dtype_bits,
              dtype_lanes, device_type,
              device_id, &outputNdarray);

/* 1. copy inference input from CPU to target
 * 2. run TVM's graph executor
 * 3. copy output back to CPU */
TVMArrayCopyFromTo(input, inputNdarray, nullptr);
set_input(input_name, inputNdarray);
run();
get_output(output_index, outputNdarray)
TVMArrayCopyFromTo(out, outputNdarray, nullptr);

/* write result pointer to output fifo
   tensor_out */
tvm::runtime::NDArray *output = &(out);
lide_c_fifo_write(tensor_out, &output);
```

on higher-level (graph-level) partitioning and scheduling is highly complementary to the intra-operator optimizations that are supported intensively by most LLCLs.

4.3.4.1 Types of Parallelism

To understand the potential utility of efficient dataflow schedule design, as supported by LCIP, it is helpful to study the relationship between different types of parallelism for a given dataflow graph, and how they affect the runtime performance of the dataflow graph. Here, we provide a brief introduction to different types of parallelism that can be utilized by LCIP. The different types of parallelism described here are summarized in Table 4.2.

Table 4.2: Summary of different types of dataflow parallelism with examples.

Dataflow Graph	Parallelism	Example	Schedule
	Pipeline		**Time Steps** T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 Subgraph G_1 — Process 1: A B A B A B A B Subgraph G_2 — Process 2: C D C D C D
	Data		**Time Steps** T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 Device 1 — Process 1: A B C D A B C D Device 2 — Process 2: A B C D A B C D
	Task		**Time Steps** T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 Process 1: A B D A B D A B Process 2: C C C

Pipeline Parallelism Pipeline parallel execution of dataflow graphs takes advantage of multithreading capability in state-of-the-art CPUs or different hardware presented in a heterogeneous environment, such as a network of edge CPUs or a mixture of CPU/GPU/accelerator devices. Pipeline parallelism is achieved by enabling the overlapping execution of multiple actors from different dataflow graph iterations on separate hardware. Exploitation of such pipeline parallelism can be facilitated by partitioning a given dataflow graph G into disjoint subgraphs $G_1, G_2, \ldots, G_n$, where each subgraph G_i has its own schedule ("subschedule") and is mapped to a distinct hardware or device.

Data Parallelism Graph-level data parallel execution creates one or more complete copies of the original dataflow graph with each copy in a separate process, and carries out independent concurrent graph executions with different inputs to achieve higher overall throughput. Graph-level data parallelism is an important special case of the pipeline parallelism concept described above. While the latency for each individual graph execution may not see any improvement, compared to when no data parallelism is applied, significant throughput improvement may be achieved by mapping each copy of the given dataflow graph to a distinct thread in a multicore CPU or to a single edge device in a network of edge devices. Data parallelism is useful, for example, if the objective of throughput optimization has higher priority compared to minimization of resource usage, such as the required pool of processors or the memory requirements.

Task Parallelism Task parallel execution takes advantage of branches presented in the given dataflow graph and executes actors presented in different branches on different hard-

ware. Both the latency and throughput of the dataflow graph execution can be improved with task parallelism as the concurrent execution on parallel branches can result in performance improvement by avoiding idle wait time when executing actors in a sequential manner. Table 4.2 presents a simple but illustrative example of task parallelism and the order of execution (schedule) when the example dataflow graph is used as the LCIP input graph.

In the context of LCIP, we refer to the three types of parallelism — pipeline, data and task parallelism — as *dataflow parallelism*. These approaches to parallel schedule design can be effective at providing improvements in CNN inference efficiency at the graph level. Intra-operator parallelism, where parallel execution is performed at the intra-actor level, is also supported naturally by LCIP through its flexible integration with ML frameworks and LLCLs. However, we do not classify intra-operator parallelism as a form of dataflow parallelism in LCIP as it optimizes computations within actors instead of at the dataflow graph or inter-actor level.

All of the types of parallelism described above, including different combinations of the different types within a given implementation, are supported by LCIP through its dataflow modeling foundations.

4.3.4.2 Graph Partitioning Algorithm

Figure. 4.3 illustrates an example of partitioning the VGG-11 dataflow graph in LCIP into three segments (i.e., into subgraphs G_1, G_2, G_3). Given the three segments at runtime, they can be executed concurrently during VGG-11 inference in a pipelined

fashion, where (in steady state) subgraph G_1 for the ith iteration of the overall dataflow graph G executes concurrently with subgraph G_2 in the $(i-1)$th iteration of G and subgraph G_3 in the $(i-2)$th iteration of G. Such pipelined configurations can provide significant improvement in inference throughput when processing large streams of input images.

LCIP includes an automatic graph partitioning function to facilitate the derivation of efficient pipelined implementations. The partitioning algorithm employed in LCIP is based upon a heuristic that was initially proposed by Lin et al. [51]. More specifically, Lin introduced the Incremental Actor Re-assignment, or IAR, algorithm to create an efficient mapping of SDF graphs to hybrid CPU/GPU platforms equipped with single- or multi-core CPUs. The objective of IAR is to optimize the throughput for a given dataflow graph, with the mapping of individual actors iteratively updated in a greedy manner. In the context of LCIP, the objective is to efficiently balance the load across all available resources.

The maximum load on any processor can be measured both statically and dynamically. Static measurement takes place by calculating the maximum complexity in the number of floating point operations (FLOPs) on a given hardware target, while dynamic measurement relies on detailed profiling results for individual actors and operators in the provided CNN across all available hardware candidates. Either static or dynamic measurement methods (or any combination) can be used as input to the IAR implementation in LCIP. The result is then a static partitioning of the graph into subgraphs, where each subgraph is mapped to a distinct CPU thread.

For incorporation into LCIP, we have developed a modified version of the IAR algorithm, called LCIP-IAR, which optimizes the mapping of actor executions onto a

range of processors by balancing the load on each processor. While the original IAR algorithm focuses on maximizing the throughput of graph execution by estimating the throughput from simulation, the `loadBalance` method used in LCIP-IAR attempts to achieve a more uniform load among available processors to ensure that a similar amount of execution time is spent on each processor, regardless of the computational power that each processor is equipped with. This is facilitated by the off-line dynamic or static profiling of individual actors on all available processors so that these measurements can be used to determine a balanced mapping of actors onto available processors.

Algorithm 2: Modified Incremental Actor Re-assignment (IAR).

input : Target CNN graph $G = (V, E)$ to be deployed
input : Set of available processes $P = \{P_1, P_2, \ldots, P_n\}$
output : Mapping of actors $v \in V$ onto P
initialize: $\forall v \in V$, $bestMap(v) = P_n$ if $t(v, P_n) < \infty$ and $bestMap(v) = p_1$ otherwise
initialize: $bestTh$ = **loadBalance**(G, $bestMap$)

Function *generateMapping* *(G, P)*
  for *v in V* **do**
    $map = bestMap, th = bestTh, p^* = bestMap(v)$;
    $Q = \{q \in P | q \neq p^*\}$;
    for *p in Q* **do**
      $map' = map - \{(v, map(v))\} \cup \{(v, p)\}$;
      th' = **loadBalance**(G, map');
      if $th' > th$ **then**
        $map = map', th = th'$;
      end
    end
    if $th > bestTh$ **then**
      $bestMap = map, bestTh = th$;
    end
  end
  return $(bestMap, bestTh)$
end

The LCIP-IAR algorithm is illustrated in Algorithm 2. Intuitively, at each main step

of the greedy approach, the *loadBalance* function in Algorithm 2 attempts to optimize the uniformity of the load. This optimization process operates by evaluating the dynamic or static load on each available processor for each candidate actor to map in the algorithm step, and selecting the mapping decision for the step in a way that results in the most uniform load for the set of actors that have been mapped so far.

Configuration of subgraph partitions in LCIP, such as those derived by the LCIP-IAR method, can be performed efficiently because graph implementation in LIDE is cleanly separated from actor implementation — as described previously, as long as the compact LIDE-based interface of actors is adhered to, graph-level scheduling configurations can be changed flexibly without any need to modify actor implementations and vice versa (actor implementations can be modified without any changes required of the schedule). Using this important property involving the orthogonality of schedule and actor implementation, designers can readily fine tune (by hand) solutions produced by LCIP-IAR, or apply alternative partitioning algorithms or hand-designed partitioning solutions.

For linear (chain-structured) layouts of actors, as in the VGG example, the system designer can either rely on manually specifying the indices of actors at the boundaries of the desired subgraphs within a partition, or rely on the LCIP-IAR algorithm to determine a graph partition automatically. These options respectively provide very efficient interfaces through which designers can experiment by hand with alternative partitioning configurations, and through which automated scheduling tools can produce partitioning results as an input to LCIP. In Section 4.4, we employ the latter approach (LCIP-IAR). Extensions to connect LCIP with more sophisticated automated partitioning and scheduling techniques is an interesting direction for future work; at the same time, the high level

Table 4.3: Summary of specifications for platforms used in the experiments.

Platform	CPU	GPU	Memory	Storage
x86 Desktop	Intel i7-11700 8-core @ 2.5GHz	Intel UHD Graphics 750†	16 GB RAM	1 TB
Raspberry Pi 4 Model B	Broadcom BCM2711 4-core @ 1.5 GHz	Broadcom VideoCore VI†	8 GB RAM	64 GB
Oneplus 7 Pro	Qualcomm Snapdragon 855 8-core @ up to 2.84 GHz	Adreno 640 GPU	12 GB RAM	256 GB

† Not used in any experiments

of abstraction at which schedule exploration is performed in LCIP makes it a practical means for the design space exploration with complex systems.

4.4 Experiment

In this section, the performance and the correctness of LCIP, our proposed dataflow-based CNN inference framework, are evaluated. We execute inference using the popular pre-trained CNN models VGG-11 [64], MobileNetV2 [65], and ResNet-34 [66] on the ImageNet [67] dataset. The experiments are performed on multiple platforms, including a desktop CPU, an Android mobile phone, and a Raspberry Pi platform. The specifications for the experiment platforms can be found in Table 4.3.

We compared the classification scores of all images between the output from VGG-11, MobileNetV2, and ResNet-34 modeled in LCIP and PyTorch. We refer to the results from PyTorch with C++ APIs as the baseline in these evaluations. The pre-trained networks are trained with the ImageNet dataset. Overall, the classification results, as compared between LCIP and the baseline networks, are the same, which helps to validate the functional correctness of LCIP.

Implementations derived using PyTorch C++ are used as baselines throughout the

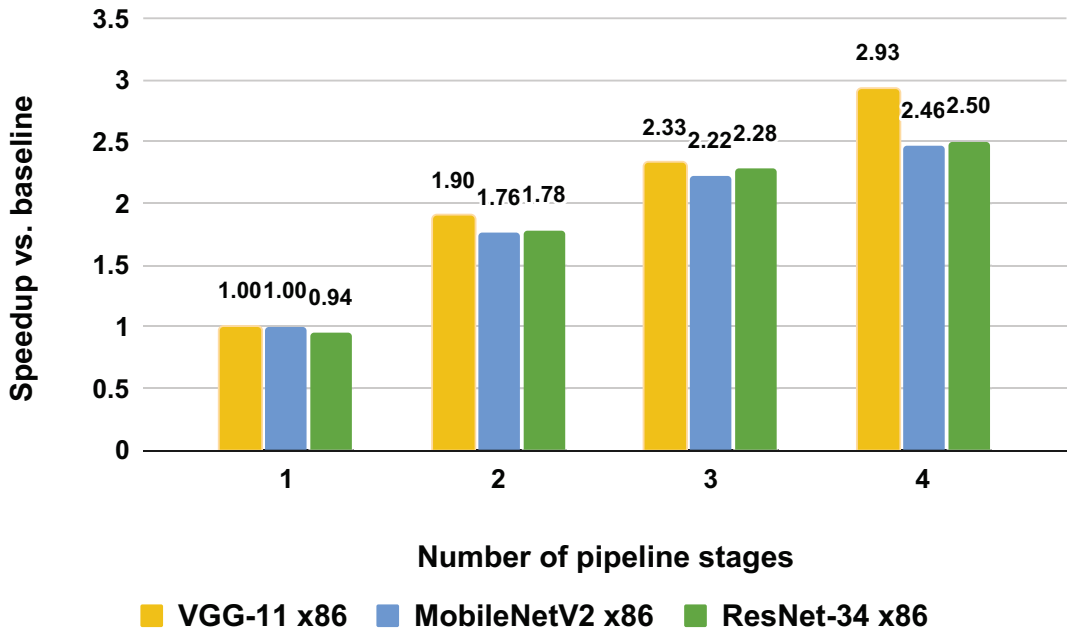


Figure 4.4: Speedup achieved by VGG-11, MobileNetV2, and ResNet-34 on x86 desktop

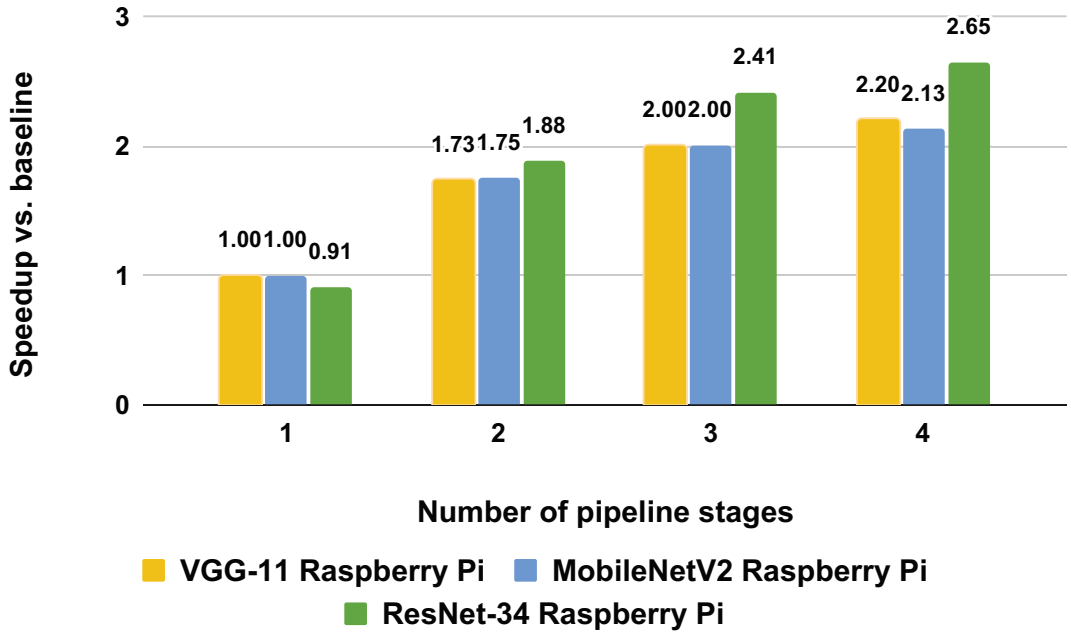


Figure 4.5: Speedup achieved by VGG-11, MobileNetV2, and ResNet-34 on Raspberry Pi

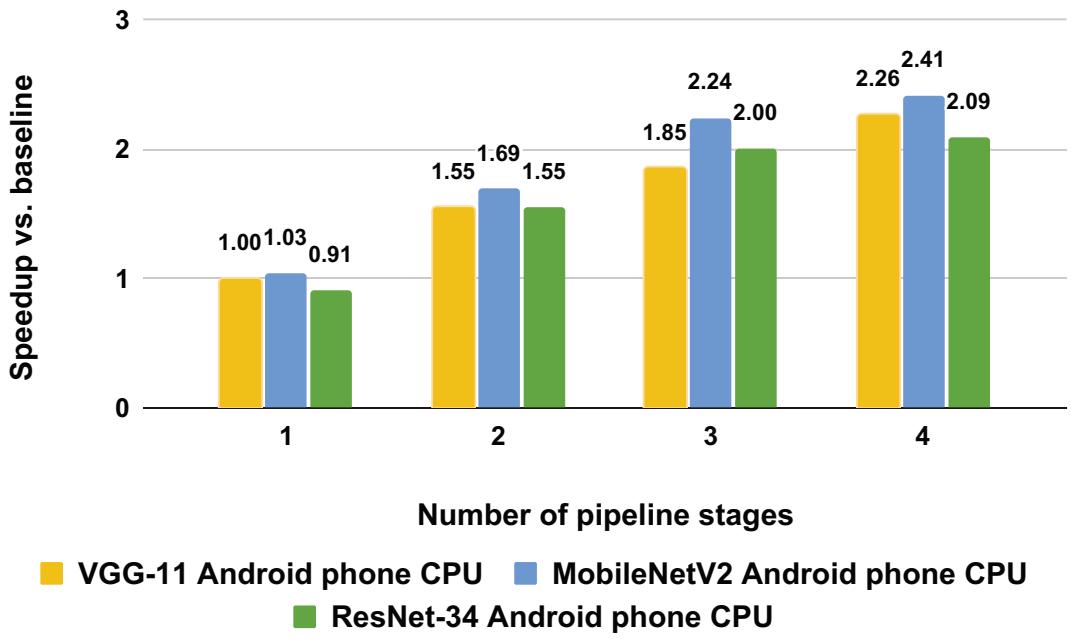


Figure 4.6: Speedup achieved by VGG-11, MobileNetV2, and ResNet-34 on Android mobile phone

Table 4.4: Results on peak memory consumption (megabytes).

	x86 CPU			Android CPU			Raspberry Pi		
Pipeline Stages	VGG-11	MobileNetV2	ResNet-34	VGG-11	MobileNetV2	ResNet-34	VGG-11	MobileNetV2	ResNet-34
1	184.76 ± 0.44	92.67 ± 0.46	279.93 ± 0.91	106.50 ± 0.16	61.85 ± 0.15	275.07 ± 0.21	106.30 ± 0.38	60.95 ± 1.03	273.80 ± 2.84
2	190.10 ± 0.44	99.09 ± 0.57	320.34 ± 1.03	111.17 ± 0.43	67.36 ± 1.26	343.39 ± 4.81	114.91 ± 0.36	71.46 ± 0.90	350.36 ± 3.23
3	199.34 ± 0.49	102.25 ± 0.66	332.57 ± 1.28	114.64 ± 0.53	78.03 ± 1.18	415.10 ± 20.10	118.13 ± 1.06	80.39 ± 1.19	438.57 ± 5.39
4	203.87 ± 2.91	109.13 ± 1.62	391.87 ± 3.19	132.55 ± 0.80	76.22 ± 3.78	447.37 ± 7.43	134.53 ± 1.00	87.13 ± 2.70	440.66 ± 5.98

experiments as we investigate inference throughput with different configurations for partitioning the input CNN. Each result is averaged over 50 runs on different platforms. The results are depicted in Figure. 4.4, Figure. 4.5, and Figure. 4.6. The throughput differences between the baselines and LCIP with no pipeline parallelism applied are negligible, as expected. As the number of pipelining stages (partitioning segments) increases, the performance reaches a measured speedup of up to 2.93×.

We also compared the memory consumption for LCIP (i.e., for the inference networks derived from LCIP) with different numbers of partitioning segments for the three

CNNs of choice. The results are shown in Table 4.4. The results include the average values and standard deviations over 50 runs for each entry in the table. Although the peak memory consumption is roughly proportional to the number of pipeline segments used in LCIP, the increase is at a slower rate than the rate of increase in inference throughput. The maximum measured increase in peak memory consumption for the experiment platforms is 1.63×. It is clear from the results that the trade-off between inference throughput and peak memory consumption is skewed towards the former, and is generally a favorable trade-off unless available memory is severely limited. This type of trade-off analysis is facilitated by the ease with which alternative dataflow parallelism configurations can be explored using LCIP.

With more recent CNNs that have reduced memory requirements, such as the MobileNet family, memory is not a major bottleneck in an increasing variety of applications for deployment at the edge. For such applications, more intensive throughput optimization can be performed as a way to leverage any increased margin for expanding CNN memory requirements.

4.4.1 Distributed and Heterogeneous Inference

We also compared the performance of LCIP-based CNN inference on a single device versus a distributed, heterogeneous platform. The devices involved in this experiment were the Raspberry Pi device and the OnePlus 7 Pro Android mobile phone; both of these devices were used in the single-device parts of the experiment and in the distributed-inference part of the experiment. Network communication in the distributed system was

implemented using Python's socket APIs, which allow binary information to be sent and received across networked devices.

In this experiment, we did not use the CPU on the Android phone for inference; we used only the GPU on that platform. Use of the GPU only in the context of the overall distributed platform helps to demonstrate the application of LCIP to a heterogeneous processing environment. The CNN model used in the experiment was MobileNetV2, which is a CNN that is relatively lean, with low computational complexity and memory footprint, making it a suitable candidate for edge-targeted deployment. Along with the dataflow parallelism exploited by LCIP, the model is further optimized in this experiment using TVM. The LCIP-IAR algorithm is used to determine the partitioning of MobileNetV2 across the Raspberry Pi and the Adreno GPU on the Android phone.

The performance of CNN inference on the single Raspberry Pi is used as the baseline in this experiment. The results are shown in Figure. 4.7. As shown, both data and pipeline parallelism for distributed inference give significant improvement in throughput, demonstrating LCIP's suitability for deploying performance-oriented CNNs on edge devices. Furthermore, it can be observed that the schedule using pipeline parallelism is outperformed by the one using data parallelism for this system. We anticipate that this is because communication across platforms under the data parallelism configuration occurs only between the host and the devices, while under pipeline parallelism, communication is also essential for passing intermediate results between the two devices. However, despite this communication overhead, the performance of the pipeline parallelism configuration is still better than using any of the two devices for MobileNetV2 inference alone.

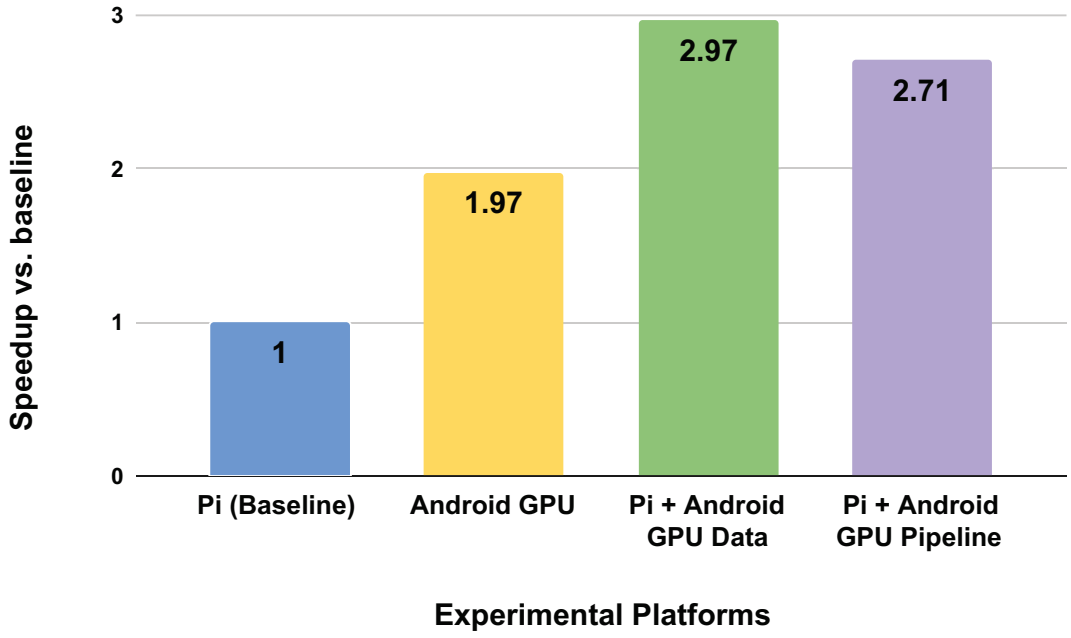


Figure 4.7: Performance comparison with relative speedup compared to the baseline with four settings: a single Raspberry Pi in isolation (the baseline), the Android phone's GPU in isolation, distributed inference with the Raspberry Pi and Android phone's GPU using graph-level data parallelism, and distributed inference with the Raspberry Pi and Android phone's GPU using pipeline parallelism. The CNN used is a TVM-optimized MobileNetV2 network.

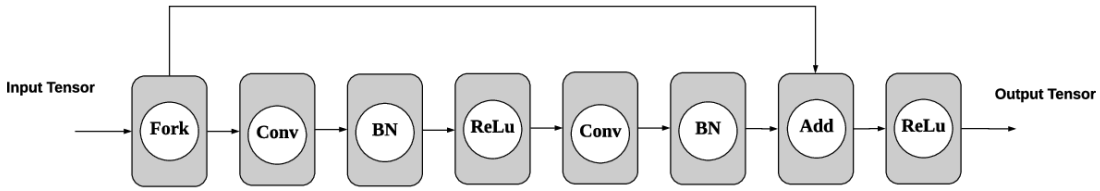


Figure 4.8: An LCIP-based dataflow model of a residual block for ResNet.

4.5 Additional Results

In this section, we provide additional details of LCIP as well as experimental results to provide more insights into how a CNN is modeled within LCIP.

4.5.1 Residual Blocks in LCIP

An LCIP-based dataflow model of a residual block for the ResNet CNN is depicted in Figure. 4.8. This model represents a fundamental component within the ResNet architecture, which was introduced by He et al. [66]. The residual block is not only a key element within ResNet but also serves as a basic building block for modeling other types of CNNs that are designed based on the ResNet architecture. LCIP provides a systematic approach to efficiently representing and executing the computations within the residual block. By utilizing LCIP, a dataflow model of the residual block can be effectively designed for edge-based CNN inference on resource-constrained devices, and such a model can be integrated into dataflow graphs for complete CNN models to facilitate high-level design analysis and optimization for the models.

Table 4.5: VGG-11 profiling results for static FLOP count and latency for convolution actors.

	Static Complexity		Raspberry Pi		X86 CPU		Android Mobile Phone	
Conv Actor Index	**FLOPs ($\times 10^6$)**	**Percentage**	**Latency (μs)**	**Percentage**	**Latency (μs)**	**Percentage**	**Latency (μs)**	**Percentage**
1	3.67	1.19%	935.37	1.45%	216.63	1.25%	290.24	1.57%
5	37.81	12.26%	5832.22	9.01%	862.01	4.98%	1294.11	6.98%
9	37.78	12.25%	5247.96	8.11%	901.47	5.21%	1317.34	7.11%
12	75.53	24.48%	10049.96	15.53%	1749.64	10.10%	2528.41	13.64%
16	37.77	12.24%	6914.79	10.68%	1429.06	8.25%	1814.71	9.79%
19	75.51	24.48%	13469.52	20.81%	2766.71	15.98%	3531.18	19.05%
23	18.88	6.12%	8348.88	12.90%	3986.85	23.03%	2541.61	13.71%
26	18.88	6.12%	8515.36	13.16%	3982.44	23.00%	2669.62	14.40%

Table 4.6: VGG-16 profiling results for static FLOP count and latency for convolution actors.

	Static Complexity		Raspberry Pi		X86 CPU		Android Mobile Phone	
Conv Actor Index	**FLOPs ($\times 10^6$)**	**Percentage**	**Latency (μs)**	**Percentage**	**Latency (μs)**	**Percentage**	**Latency (μs)**	**Percentage**
1	3.67	0.58%	1627.71	1.25%	218.26	0.75%	289.12	0.86%
4	75.63	12.00%	13200.35	10.16%	1408.76	4.86%	3062.22	9.15%
8	37.81	6.00%	5668.63	4.36%	860.79	2.97%	1313.73	3.93%
11	75.56	11.99%	10943.36	8.42%	1394.96	4.82%	2485.34	7.43%
15	37.78	6.00%	5251.72	4.04%	906.56	3.13%	1323.79	3.96%
18	75.53	11.99%	10149.24	7.81%	1737.79	6.00%	2520.76	7.53%
21	75.53	11.99%	10149.28	7.81%	1750.29	6.04%	2520.50	7.53%
25	37.77	5.99%	7014.56	5.40%	1411.62	4.87%	1819.60	5.44%
28	75.51	11.99%	13655.26	10.51%	2761.85	9.54%	3519.97	10.52%
31	75.51	11.99%	13655.64	10.51%	2785.95	9.62%	3522.49	10.53%
35	18.88	3.00%	8587.88	6.61%	3989.89	13.78%	2537.34	7.58%
38	18.88	3.00%	8571.32	6.60%	3978.42	13.74%	2542.59	7.60%
41	18.88	3.00%	9142.63	7.04%	3978.86	13.74%	2706.79	8.09%

4.5.2 Performance of LCIP on the CIFAR-10 Dataset

We compared the classification scores of all images between the output from VGG-11, VGG-16, and ResNet-34 modeled in LCIP and PyTorch. We refer to the results from Pytorch with C++ APIs as the baseline in these evaluations. The pre-trained networks are trained with the CIFAR-10 dataset. Figure. 4.9 shows the classification scores of selected images from CIFAR-10. Overall, the classification results between LCIP and the baseline networks are the same, which helps to validate the functional correctness of LCIP.

Figure 4.9: Classification scores on selected output images from the VGG-16 network modeled and implemented using LCIP.

Table 4.7: ResNet-34 profiling results for static FLOP count and latency for convolution actors.

	Static Complexity		Raspberry Pi		X86 CPU		Android Mobile Phone	
Conv Actor Index	**FLOPs ($\times 10^6$)**	**Percentage**	**Latency (μs)**	**Percentage**	**Latency (μs)**	**Percentage**	**Latency (μs)**	**Percentage**
1	3.54	0.61%	879.93	0.26%	214.41	0.37%	267.99	0.28%
5	18.87	3.23%	11717.43	3.43%	1402.41	2.43%	3092.18	3.26%
8	18.87	3.23%	11673.26	3.42%	1339.68	2.33%	3118.52	3.28%
13	18.87	3.23%	11734.23	3.44%	1334.01	2.32%	3137.05	3.30%
16	18.87	3.23%	11675.63	3.42%	1332.35	2.31%	3125.58	3.29%
21	18.87	3.23%	11785.66	3.45%	1334.68	2.32%	3132.84	3.30%
24	18.87	3.23%	11791.72	3.45%	1335.13	2.32%	3152.22	3.32%
29	9.44	1.62%	5882.48	1.72%	805.92	1.40%	1737.02	1.83%
32	18.87	3.23%	9804.64	2.87%	1393.84	2.42%	2480.39	2.61%
34	1.05	0.18%	956.25	0.28%	210.66	0.37%	324.03	0.34%
39	18.87	3.23%	9826.89	2.88%	1368.30	2.38%	2472.94	2.60%
42	18.87	3.23%	9810.66	2.87%	1367.90	2.37%	2477.84	2.61%
47	18.87	3.23%	9818.94	2.88%	1368.19	2.38%	2486.70	2.62%
50	18.87	3.23%	9796.57	2.87%	1363.60	2.37%	2484.15	2.62%
55	18.87	3.23%	9811.45	2.87%	1361.86	2.36%	2487.88	2.62%
58	18.87	3.23%	9800.45	2.87%	1360.24	2.36%	2492.87	2.63%
63	9.44	1.62%	5036.02	1.48%	992.00	1.72%	1468.50	1.55%
66	18.87	3.23%	9179.48	2.69%	1784.82	3.10%	2502.30	2.64%
68	1.05	0.18%	837.75	0.25%	227.64	0.40%	272.27	0.29%
73	18.87	3.23%	9186.75	2.69%	1743.61	3.03%	2486.58	2.62%
76	18.87	3.23%	9170.47	2.69%	1738.16	3.02%	2499.93	2.63%
81	18.87	3.23%	9190.96	2.69%	1736.34	3.01%	2505.69	2.64%
84	18.87	3.23%	9158.97	2.68%	1734.65	3.01%	2507.09	2.64%
89	18.87	3.23%	9174.72	2.69%	1732.71	3.01%	2510.19	2.64%
92	18.87	3.23%	9154.22	2.68%	1730.43	3.00%	2508.46	2.64%
97	18.87	3.23%	9159.91	2.68%	1731.27	3.01%	2509.35	2.64%
100	18.87	3.23%	9148.05	2.68%	1731.69	3.01%	2507.94	2.64%
105	18.87	3.23%	9166.77	2.69%	1732.48	3.01%	2510.76	2.64%
108	18.87	3.23%	9169.13	2.69%	1734.91	3.01%	2515.26	2.65%
113	9.44	1.62%	6279.39	1.84%	1407.93	2.44%	1859.36	1.96%
116	18.87	3.23%	12309.01	3.61%	2748.61	4.77%	3504.05	3.69%
118	1.05	0.18%	935.92	0.27%	242.85	0.42%	329.68	0.35%
123	18.87	3.23%	12302.23	3.60%	2730.40	4.74%	3483.36	3.67%
126	18.87	3.23%	12307.68	3.61%	2763.57	4.80%	3512.34	3.70%
131	18.87	3.23%	12321.37	3.61%	2764.96	4.80%	3517.29	3.70%
134	18.87	3.23%	12323.92	3.61%	2767.88	4.81%	3523.91	3.71%

4.5.3 CNN Profiling Results and Performance

Table 4.5, Table 4.6, and Table 4.7 show the Floating Point Operation (FLOP) count, as well as platform-specific latency measurements for the convolution (Conv) actor for the VGG-11, VGG-16, and ResNet-34 CNNs. These results are derived automatically through profiling utilities that are available as part of LCIP. Such results are useful in informing hand-driven or automated partitioning processes for platform-specific design optimization. The profiling results shown in these three tables include the percentage of the individual-actor FLOP count and latency in relation to the overall network. By adding up the values in each percentage column, it becomes clear that convolution is overwhelmingly the dominating actor-type in terms of FLOP count and latency.

The X86 CPU used in our experiments is an Intel Core i7-2600K-3.40GHz with 4 cores. Choosing CNN inference with PyTorch C++ as the baseline, we investigate the throughput with different partition settings. The results averaged over 50 runs are depicted in Figure. 4.10. The throughput difference between the baseline and our dataflow CNN inference framework with no pipeline parallelism applied is negligible. As the number of pipelining stages (partitioning segments) increases, the performance reaches a measured speedup of up to 2.93×.

The Android mobile phone used in our experiments is the OnePlus 7 Pro, equipped with an 8-core Qualcomm CPU, 256 GB storage, and 12 GB of RAM. The Raspberry Pi is equipped with a 4-core CPU, 64GB storage, and 8GB of RAM. Similar to the LCIP performance on the X86 CPU compared to the baseline, LCIP consistently outperforms the PyTorch C++ baseline on these two resource-constrained devices for the three CNNs.

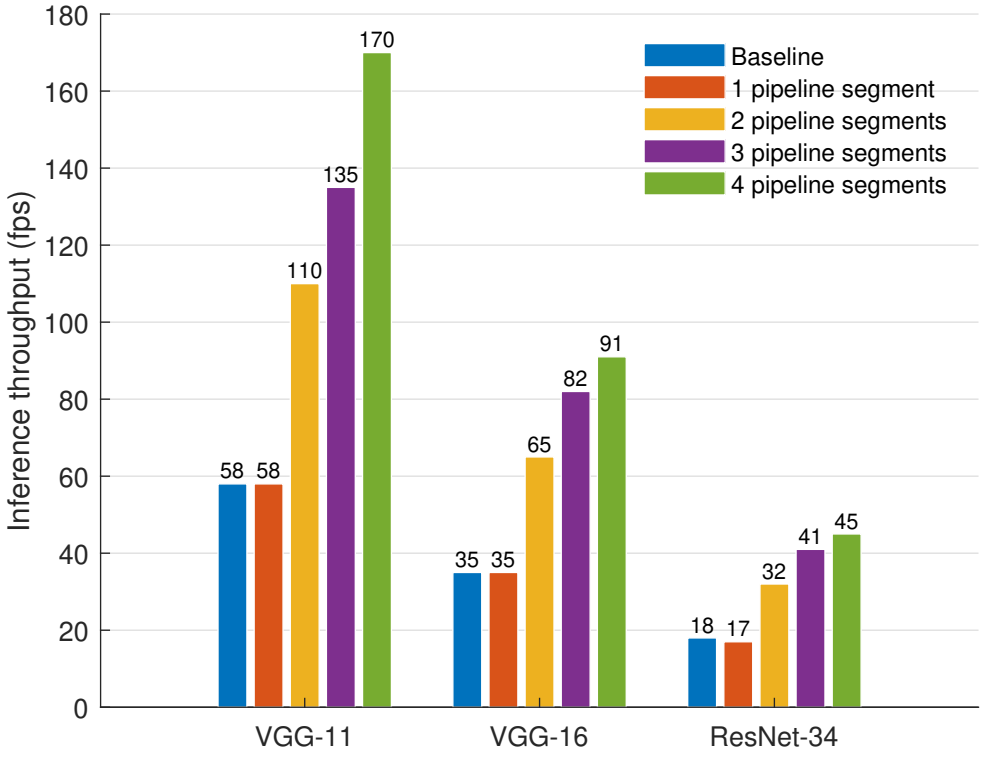


Figure 4.10: Performance comparison using an X86-CPU-based desktop platform to execute inference by the three evaluated networks: VGG-11, VGG-16, and ResNet-34.

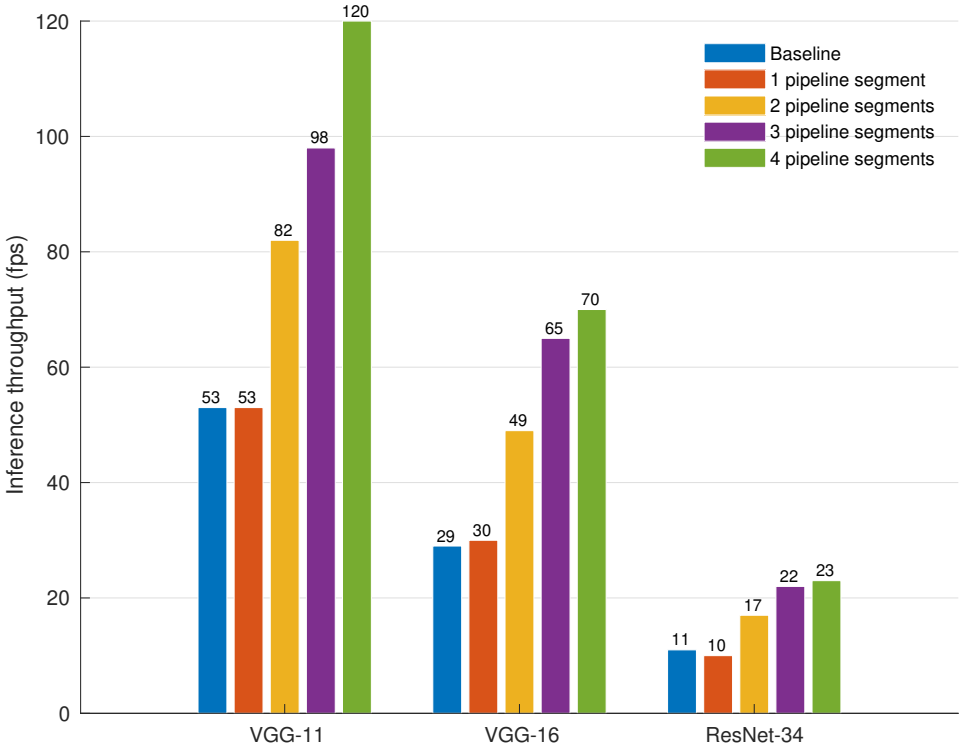


Figure 4.11: Performance comparison using an Android mobile phone platform to execute inference by the three evaluated networks: VGG-11, VGG-16, and ResNet-34.

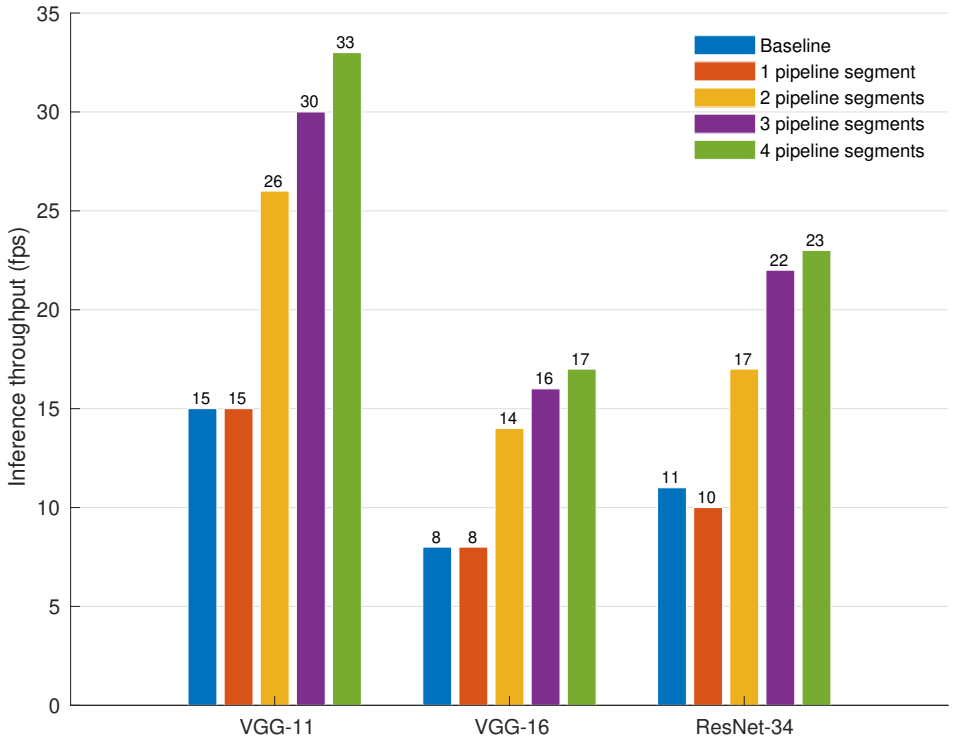


Figure 4.12: Performance comparison using a Raspberry Pi platform to execute inference by the three evaluated networks: VGG-11, VGG-16, and ResNet-34.

The maximum inference-throughput speedup is seen to be at least 2.22× on the Android mobile phone and 2.11× on the Raspberry Pi, as shown in Figure. 4.11 and Figure. 4.12.

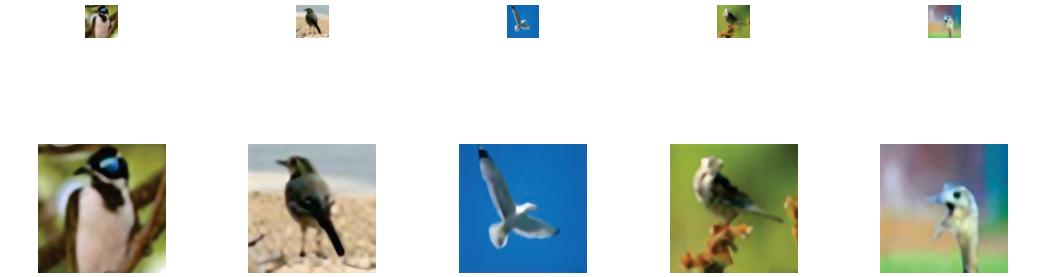
4.5.4 Image Super Resolution

4.5.4.1 Introduction

Image super-resolution is applied in various application areas — for example, for medical imaging, image super-resolution techniques are used to enahnce the quality of medical images to facilitate diagnostic processes that use the images as input. For network communications, image super resolution also serves an important role, as it allows images to be communicated with low resolution and then enhanced after being received into forms that have higher image quality.

CNN-based approaches have been widely studied for image-super resolution applications (e.g., see [68, 69, 70]). However, because of the high computational complexity introduced by the convolution operation, the key operation in CNNs, developing and deploying efficient image super-resolution systems with CNNs incorporated can be challenging. Therefore, there is a need for systematic approaches to design and implementation of efficient image super-resolution systems.

In this section, LCIP is further demonstrated by applying it to a DCNN that is designed for image super-resolution.



ce752b19084ed397 b1338d4d3609c6f6 c33c36c71c1de32c f146dccf1c3218d3 d47a2d1696a919e5

Figure 4.13: Original images from the CIFAR-10 dataset and images with resolution enhanced by SRResNet with their perceptual hash values.

4.5.4.2 Related Work

In this section, we discuss related work on CNN models for image super-resolution. SRCNN provides a novel deep learning based method for single-image super-resolution problems [70]. Although the CNN proposed in [70] is simple in terms of network complexity, it is shown to be effective compared to alternative image super-resolution approaches. Shortly after the introduction of SRCNN, another CNN based method, VDSR [71], was introduced. VDSR was shown to provide much better performance in terms of PSNR than SRCNN. VDSR advances the use of CNNs in image super resolution by designing a VGG-inspired CNN to increase accuracy and visual improvement. Other attempts at improving the accuracy of image super-resolution made use of ResNet-inspired CNN architectures, such as EDSR [72] and SRResNet [68]. Both of these works utilized the residual block as their network's key building block to further enhance network performance.

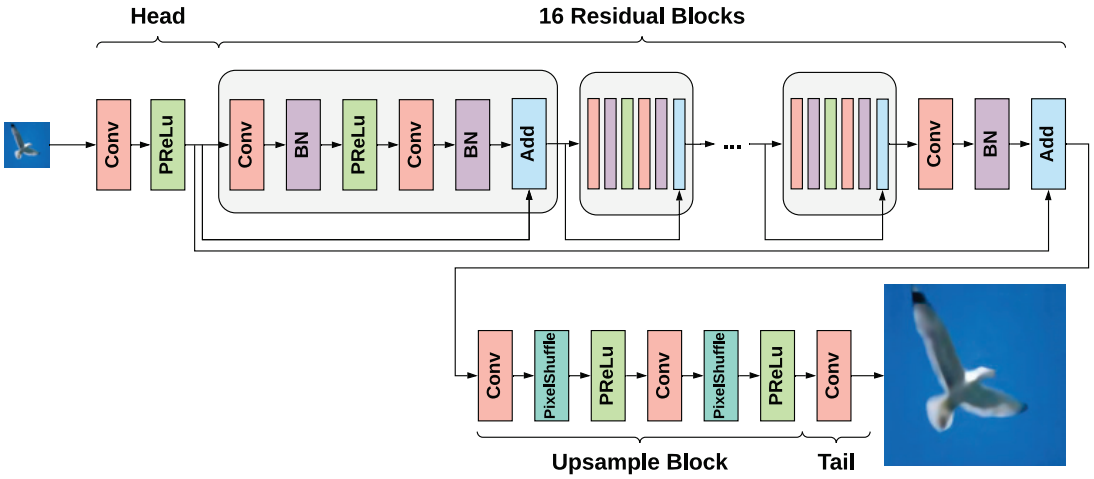


Figure 4.14: Network architecture of SRResNet.

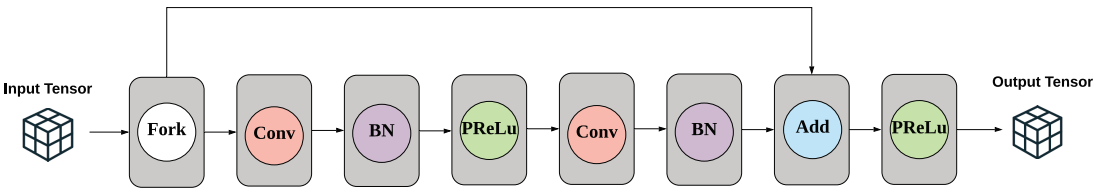


Figure 4.15: Residual block for SRResNet modeled in LCIP.

4.5.4.3 Results on Image Super Resolution with SRResNet

We also present experimental results on the SRResNet [68] network modeled with LCIP for image resolution enhancement on the target devices listed previously. The input to the network is from the CIFAR-10 dataset with resolution 32×32. The network is pretrained with weights obtained from [73]. The scaling factor of SRResNet used in our experiments is 4.

In Figure. 4.13, the top row indicates selected images from the CIFAR-10 dataset that are used as inputs to the SRResNet network. The middle row shows the resulting images, where the resolution of the resulting images has been enhanced by a factor of 16 — from 32×32 to 128×128, as the scaling factor used for this particular network is 4. The bottom row in Figure. 4.13 contains the perceptual hashing [74] values of the output images as they are used to verify the correctness of SRResNet modeled in LCIP. We applied the perceptual hash on all output images from the original SRResNet as well as from SRResNet modeled in LCIP, and we found no discrepancy between the hashing outcomes.

The network architecture of SRResNet and the modeling of the residual block for SRResNet in LCIP are shown in Figure. 4.14 and Figure. 4.15, respectively. SRResNet from [68] consists of four sections, as indicated in Figure. 4.14: head, residual blocks, upsample block, and tail. The main difference between SRResNet and the original ResNet is that there is a skip connection connecting the beginning of the 16 residual blocks and the end of those residual blocks. Most actors are identical to the actors used in our LCIP-based modeling of ResNet-34, except for the PReLu actor, PixelShuffle actor, and a hy-

Table 4.8: SRResNet profiling results for static FLOPs count and latency for each convolution actors measured on X86 CPU, Android mobile phone, and Raspberry Pi

	Static Complexity		**Raspberry Pi**		**X86 CPU**		**Android Mobile Phone**	
Conv Actor Index	**FLOPs (×106)**	**Percentage**	**Latency (μs)**	**Percentage**	**Latency (μs)**	**Percentage**	**Latency (μs)**	**Percentage**
1	32	0.70%	5859.03	0.31%	1517.23	0.91%	1350.35	0.31%
5	76	1.66%	13026.26	0.70%	1377.51	0.82%	3088.50	0.70%
8	76	1.66%	12838.09	0.69%	1344.56	0.80%	3108.45	0.71%
12	76	1.66%	12908.67	0.69%	1341.40	0.80%	3116.80	0.71%
15	76	1.66%	12830.36	0.69%	1344.09	0.80%	3108.15	0.71%
19	76	1.66%	12899.60	0.69%	1341.23	0.80%	3116.76	0.71%
22	76	1.66%	12797.29	0.68%	1343.96	0.80%	3105.30	0.71%
26	76	1.66%	12870.64	0.69%	1342.15	0.80%	3113.13	0.71%
29	76	1.66%	12762.91	0.68%	1348.52	0.81%	3104.76	0.71%
33	76	1.66%	12851.84	0.69%	1347.45	0.81%	3112.31	0.71%
36	76	1.66%	12760.30	0.68%	1349.68	0.81%	3103.30	0.71%
40	76	1.66%	12845.69	0.69%	1345.24	0.80%	3111.70	0.71%
43	76	1.66%	12754.48	0.68%	1348.54	0.81%	3104.55	0.71%
47	76	1.66%	12840.94	0.69%	1346.75	0.80%	3110.27	0.71%
50	76	1.66%	12747.75	0.68%	1352.04	0.81%	3100.67	0.71%
54	76	1.66%	12842.37	0.69%	1348.94	0.81%	3109.83	0.71%
57	76	1.66%	12747.15	0.68%	1352.60	0.81%	3100.30	0.71%
61	76	1.66%	12837.12	0.69%	1348.58	0.81%	3107.84	0.71%
64	76	1.66%	12748.40	0.68%	1351.04	0.81%	3099.72	0.71%
68	76	1.66%	12835.55	0.69%	1348.25	0.81%	3107.84	0.71%
71	76	1.66%	12745.39	0.68%	1349.31	0.81%	3097.95	0.71%
75	76	1.66%	12833.59	0.69%	1346.47	0.80%	3101.85	0.71%
78	76	1.66%	12743.84	0.68%	1348.70	0.81%	3093.80	0.71%
82	76	1.66%	12834.97	0.69%	1346.87	0.80%	3102.98	0.71%
85	76	1.66%	12735.79	0.68%	1349.78	0.81%	3094.82	0.71%
89	76	1.66%	12819.50	0.68%	1346.83	0.80%	3100.04	0.71%
92	76	1.66%	12727.98	0.68%	1349.22	0.81%	3087.69	0.70%
96	76	1.66%	12817.38	0.68%	1346.72	0.80%	3098.39	0.71%
99	76	1.66%	12729.22	0.68%	1349.75	0.81%	3090.59	0.70%
103	76	1.66%	12815.23	0.68%	1347.16	0.81%	3097.24	0.71%
106	76	1.66%	12734.65	0.68%	1350.92	0.81%	3088.19	0.70%
110	76	1.66%	12815.21	0.68%	1346.77	0.80%	3096.68	0.71%
113	76	1.66%	12722.37	0.68%	1349.91	0.81%	3088.95	0.70%
116	76	1.66%	12816.75	0.68%	1346.33	0.80%	3096.53	0.71%
119	302	6.61%	35720.38	1.91%	5063.35	3.03%	9236.41	2.11%
122	1210	26.48%	139392.95	7.45%	20029.31	11.97%	37718.30	8.60%
125	510	11.16%	1152978.85	61.58%	82739.32	49.45%	240835.34	54.91%

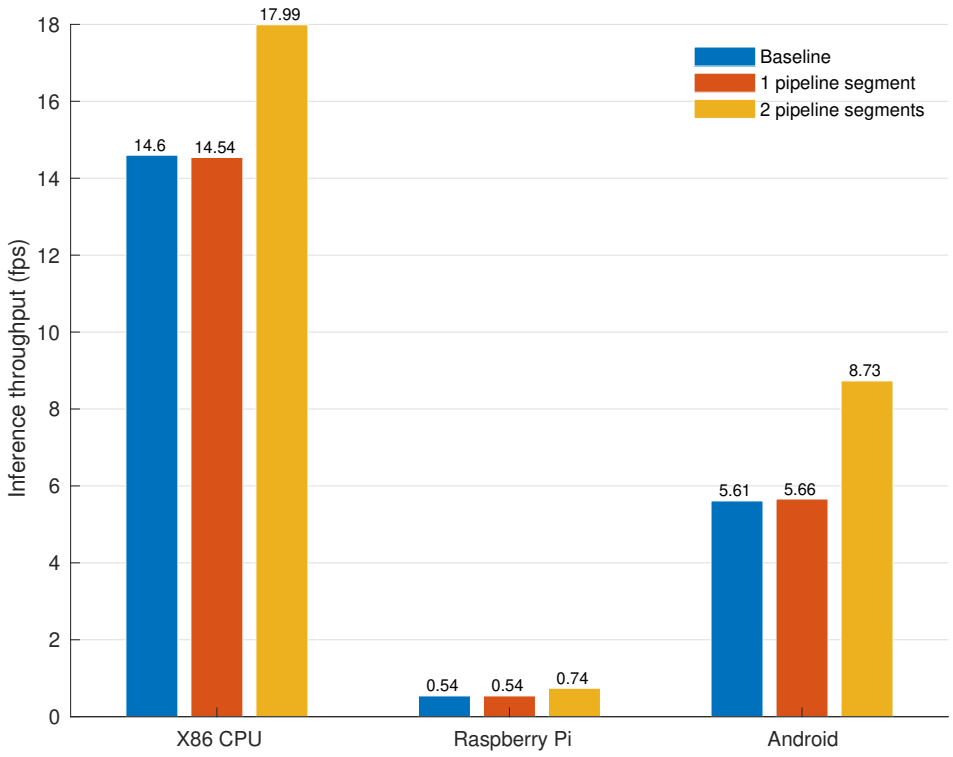


Figure 4.16: Performance comparison of SRResNet on X86 CPU, Raspberry Pi, and Android mobile phone.

perbolic tangent actor at the end of the network.

Table 4.8 shows information on the static and measured complexity of SRResNet, similar to Table 4.5, Table 4.6, and Table 4.7. The particular network we use in the experiments has 16 residual blocks between the head block and upsample block. The measured latency of the last convolution actor shows that around 50% of the computation happens in that specific actor, which lead us to choose the partition point for the SRResNet in LCIP to be at that specific actor. Choosing the partition point in this way helps to balance the load between threads when the graph is partitioned.

Figure. 4.16 shows that the LCIP version of SRResNet achieves similar perfor-

Table 4.9: Results of SRResNet on peak memory consumption (megabytes) from our experimentation with LCIP on the three targeted devices.

Segments \ Device	X86		Raspberry Pi		Android	
	Avg.	Std.	Avg.	Std.	Avg.	Std.
1	270.03	2.90	1,080.07	3.74	1,075.69	0.21
2	336.81	13.53	1,279.83	16.03	1,188.48	17.64

mance in terms of inference throughput compared to the PyTorch baseline in C++. However, LCIP outperforms the baseline with more pipeline segments enabled. The speedup results for the X86 CPU, Raspberry Pi, and Android mobile phone with 2 pipeline segments are 1.24x, 1.37x, and 1.54x, respectively. Increasing the number of pipeline segments to 3 did not provide any speedup in our experiments; we anticipate that this is mainly due to the high computational complexity in the last convolutional layers. To achieve additional increase in inference throughput, it may be useful to explore further optimization in the low-level compute library. Any such optimization in the low-level library can be readily leveraged by LCIP.

We also compared the peak memory consumption for SRResNet implemented with LCIP on the three target devices. Our measured results on the peak memory consumption of SRResNet with LCIP can be found in Table 4.9. We present the average peak memory consumption and the standard deviation in the units of Megabytes on the three target devices: X86 CPU, Raspberry Pi, and Android mobile phone. From Table 4.9, we can see that the X86 CPU suffers the most from increasing number of pipeline stages with a 1.31x increase in peak memory consumption, a relatively large factor of increase compared to the factor of throughput increase that is achieved from pipelining. For the Raspberry Pi and Android mobile phone, however, the peak memory usage increase-factor is lower than

the inference throughput increase-factor as the number of pipeline segments increases.

4.6 Conclusion

In this chapter, we have presented a software tool, called the Lightweight-dataflow-based CNN inference package (LCIP), for streamlined implementation of deep neural networks (DNNs) on resource-constrained platforms. LCIP addresses in an integrated manner the issues of retargetability, resourceability, efficiency, and configurability, which are all important in edge-based deployment of DNNs. Through extensive experimental evaluation on three state-of-the-art networks for image classification, we have demonstrated the effectiveness of LCIP in a heterogeneous, distributed edge device network. Useful directions for future work include extension of LCIP beyond C/C++ (e.g., to include implementations that are based on hardware description languages), experimentation with LCIP on platforms beyond CPUs and GPUs, such as hardware accelerators, and integrating LCIP with additional methods for automated dataflow graph partitioning and scheduling.

Chapter 5

Multi-Objective Design Optimization for Image Classification Using Elastic Neural Networks

In Chapter 3, we examined how varying network input complexity and network topology affect the accuracy, throughput, and peak memory consumption of neural networks deployed on resource-constrained platforms, specifically for hyperspectral image classification problems. Building on top of these results, our objective in this chapter is to expand our investigation and explore systematic trade-off optimization for neural network deployment on resource-constrained platforms. This exploration explicitly considers a multi-dimensional design space, including accuracy and runtime performance metrics. Our exploration in this chapter also extends beyond specific use cases, and provides a methodology that can be applied to a wide variety of application scenarios.

In this chapter, we address multi-objective optimization for CNN implementation, and we explore techniques for configuring (statically or dynamically) CNN network topologies for arbitrary CNNs. To this end, we introduce a novel Pareto-front-based early-exit selection algorithm. By combining insights from Chapter 3 and Chapter 4, and the current chapter, we establish an end-to-end systematic approach for implementing CNNs on resource-constrained platforms.

Material in this chapter was published in partial, preliminary form in [75].

5.1 Introduction

Deep convolutional neural networks (DCNNs) are among the most popular state-of-the-art approaches in visual object recognition tasks. DCNNs have been extensively applied in classification tasks since the *ILSVRC* challenge [76]. During the history of this competition, a major sustained trend has been the increase in network depth: for example, AlexNet [77] with 8 layers won first place in 2012 with a top-5 error rate of 16%; VGG [64], consisting of 16 layers, won first place and decreased the error rate to 7.3% in 2014; and ResNet [66], with 152 very deep convolutional layers and identity connections, won first place while further decreasing the error rate to 3.6% in 2015.

The evolution of networks such as AlexNet, VGG, and ResNet has thus demonstrated progressively decreasing error rates with corresponding increases in network complexity. For example, AlexNet has approximately 0.33 GFLOPs (Floating Point Operations), VGG-16 has around 7.69 GFLOPs, and ResNet-152 has around 5.79 GFLOPs with optimized complexity compared to VGG-16. Due to their rapidly escalating complexity, DCNN-based inference has become an increasing challenge for real-time image classification applications.

The methodology of elastic neural networks provides an approach to addressing this challenge by equipping DCNNs with multiple exit points that can be used to deliver alternative trade-offs between accuracy and real-time performance [78, 79]. Once an elastic network is deployed, the exit point to use can be determined in a manner that is specific to the deployment and it can also be readily adjusted dynamically as the network operates. The particular exit point to choose for a given deployment or operational context can de-

pend on factors such as the specific real-time constraints of the application, the available computational resources, the available energy budget for neural network processing, and the requirements of the network in terms of accuracy. In this chapter, we propose a novel framework for selecting early exit points when converting a DCNN into an elastic neural network. Our approach systematically considers the *diversity* of trade-offs delivered by the set of selected exit points. A diverse set of trade-offs is in general more useful across a range of deployment scenarios than a set of trade-off points that are clustered around the same point in the given multidimensional design evaluation space (e.g., the space involving accuracy and execution time). The core novelty of our proposed approach is the integration of concepts related to multi-objective optimization and Pareto diversity into the emerging framework of system design for elastic neural networks.

Multi-objective optimization has been studied widely in previous works (e.g., see [80, 81, 82]). Jin and Sendhoff provide an overview of Pareto-based multi-objective optimization approaches used in machine learning [80]. They show that Pareto-based multi-objective learning approaches are more effective than learning algorithms that have scalar cost functions. The types of machine learning algorithms that can benefit from Pareto-based multi-objective learning approaches include clustering, feature selection, and knowledge extraction. Compared to using scalar cost functions in machine learning algorithms, multi-objective learning approaches can enhance the system design process by taking into account Pareto fronts in multidimensional design evaluation spaces, such as spaces involving inference accuracy, throughput, and memory requirements. Multi-objective optimization approaches have been studied to navigate trade-offs between training accuracy and model complexity achieved by alternative neural network models for the

same inference problem [83]. Chen et al. discussed techniques to select Pareto-optimal points out of all available points for multi-objective problems [81].

Using early exits provides a general way to transform deep neural network models into adaptive computational graphs, as demonstrated in [84, 85, 78, 79]. In this chapter, we generalize the elastic neural network design approach proposed in [79] to encompass a wider range of DCNNs, and to systematically take into account the diversity of trade-offs offered by different candidate sets of exit points for an elastic neural network implementation. To the best of our knowledge, there is no previous study on applying multi-objective optimization towards the problem of selecting exit-sets for elastic neural networks.

In the context of machine learning system design, evaluating the quality of a given set of design points is a useful topic to study because many real-world system design problems do not have unique optimal solutions associated with them — instead, they are characterized by trade-offs, which make some solutions more attractive than others depending on specific constraints and objectives associated with how the systems are being deployed or how they are being operated at a particular point in time.

The approach proposed in this chapter incorporates a quality indicator for evaluating collections of alternative design points in terms of the diversity and coverage of their associated trade-offs. The quality indicator, called the diversity comparison indicator (DCI), assesses the relative quality of a set of Pareto points based on their uniformity and spread across a given multi-dimensional design evaluation space [86]. The DCI is useful because it is computationally efficient, and it overcomes certain limitations of other diversity assessment metrics for design evaluation spaces. In particular, related diversity assessment metrics fail to assess arbitrary numbers of Pareto sets, due to the need for

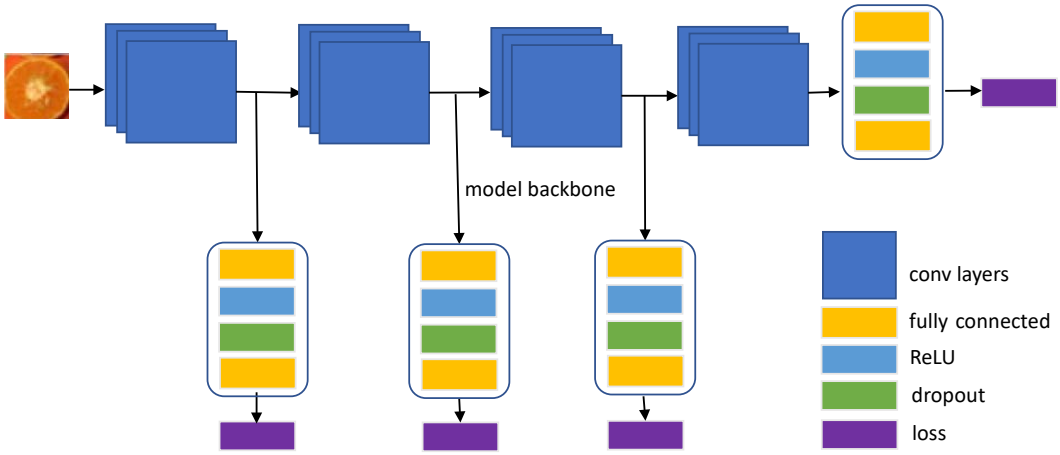


Figure 5.1: The architecture of the proposed method.

predetermined references in order to compare qualities of sets, as seen in [87, 88]. Additionally, the performance of existing diversity assessment metrics degrades when the number of objectives increases [89].

5.2 Approach

The architecture of the proposed elastic neural network framework is shown in Figure. 5.1. The methodology used to design such an architecture involves adding additional layers, called early-exit layers, to a given pre-trained DCNN for a given classification task. The early-exit layers are trained separately from the main part of the network. During the training process for the early-exit layers, the weights of the main part of the network remain unchanged, only the weights within the early-exit layers are updated.

In our approach to configuring an elastic DCNN N, early exits are added to the pre-trained backbone network and trained separately to achieve maximum possible accuracy at different depths of a given network. Once the layers associated with each early exit

point are trained, a lookup table T is generated to record the trade-offs associated with all of the available early exit options. A subsequent objective in our approach is to analyze this lookup table to extract a compact subset of early exits that will actually be used to implement the network.

More precisely, our objective is to develop a transformation Γ that takes as input a DCNN N with available early exits $Z(N) = \{E_1, E_2, \ldots, E_m\}$, where $i < j$ whenever E_i is closer to the network input (at a shallower depth in the network) compared to E_j. Γ also takes as input a positive integer $k < m$, which gives the number of early exits to include in the transformed network. When implementing a large network N on an edge device, typically $k \ll m$ to avoid excessive memory requirements due to the k early-exit layers that need to be added to the elastic DCNN.

The output $\Gamma(N, k)$ is a k-element array A of increasing positive integers in $[1, m]$, which correspond to the indices of the early exits to implement in the network corresponding to the transformed output $\Gamma(N, k)$. In particular, the transformed network corresponding to $\Gamma(N, k)$ consists of the early exits $E_{A[1]}, E_{A[1]}, \ldots, E_{A[k]}$ together with the k early exit layers that are needed to derive classification outputs from each of the k selected intermediate points in the network.

The lookup T table is defined such that $T \in \mathbb{R}^{m \times n}$, where $\mathbb{R}$ denotes the set of real numbers, m is the number of available early exits, and n is the number of relevant implementation metrics. Examples of metrics that may be relevant for this purpose, depending on the design context, are inference throughput, latency, energy consumption (e.g., an estimate of the average energy consumed per inference operation), memory requirements (e.g., for parameter storage), and classification accuracy.

By convention, the metrics associated with T are defined such that higher values correspond to worse performance or quality compared to lower values. For example, the *accuracy* metric is commonly used to measure the performance of a DCNN for a specific task, such as classification and object detection. Clearly, higher values indicate better quality for a network with respect to this metric. To apply accuracy as a metric in the lookup table T, we convert it to a lower-is-better metric. In particular, we assume that the accuracy α is measured as a percentage value in $[0, 100]$, and we define the corresponding metric for T as $(100 - \alpha)$, which can be viewed as the error rate.

In the table T, each ith row corresponds to the values for the n relevant metrics associated with early exit E_i; there are n entries per row. Each column of T contains values associated with a specific metric collected from all m of the early exits. Thus each $T[i, j]$ gives the value of metric j that is associated with early exit E_i. The $m \times n$ elements of the table T can be populated by simulating the network with different early exit settings or executing the network on the target platform, and collecting the measured results for each metric.

The size of the table T can be reduced by excluding points (early exit candidates) that are *dominated* in terms of multi-objective optimization. Dominance in our context is defined as follows: early exit E_a is dominated by E_b if: (1) there is at least one metric p such that $T[a, p] > T[b, p]$, *and* (2) for every other metric q, we have that $T[a, q] \geq T[b, q]$. Dominated points can be viewed as redundant or useless in relation to the given set of relevant metrics and alternative design points.

Dominated points (early exit candidates) can be filtered out from T through a simple process of pairwise comparison between distinct rows of T, where each comparison

between a pair of rows involves comparing corresponding columns of T; this can be done in $O(m^2 n)$ time. As soon as an early-exit candidate x is found to be dominated by another, we can remove x from T (i.e., remove the row corresponding to x) and leave this row out of any further comparisons. We denote the number of rows remaining in T after removing all dominated rows by m'. The dimensions of T after removing the dominated rows is thus, $m' \times n$, where $m' \leq m$ and equality $m' = m$ holds when all members of $Z(N)$ are non-dominated (i.e., when all are Pareto points to begin with).

Note that we use a minor abuse of notation where T is used to denote both the original $(m \times n)$ lookup table of network exit points, and the table consisting only of non-dominated points. In the remainder of this chapter, we assume the latter meaning — the table of non-dominated early exit candidates, unless otherwise stated.

The rows $i = 1, 2, \ldots, m'$ in T correspond to a set of early exits $Y(N) = \{F_1, F_2, \ldots F_{m'}\}$, where $Y(N) \subseteq Z(N)$. For each $F_i \in Y(N)$, we denote the corresponding member $E_j \in Z(N)$ by $H(F_i)$.

A key aspect in the novelty of our proposed elastic DCNN approach is the integration of the DCI assessment metric described in Section 5.1 to enhance the adaptivity of derived DCNNs across a wide range of operational constraints and objectives. Intuitively, the DCI provides a consistent metric to compare the diversity of multiple sets of Pareto points in a multi-objective optimization problem. Evaluation of this metric involves mapping the Pareto points into an n-dimensional grid, and determining the contribution that each set of points has to each cell (hyperbox) in the grid. For further details on how the DCI is formulated and computed, we refer the reader to [86].

Algorithm 3 utilizes a greedy approach to incrementally construct the set of selected

exits $\Gamma(N, k)$ for the elastic DCNN that is derived as part of the proposed design methodology. When the algorithm completes, the final value of the set β gives the value $\Gamma(N, k)$ that is to be used to provide the exit points in the transformed elastic DNN.

The function *selectExit* encapsulates DCI index computation, and is used to compare exit point candidates to determine the best one in terms of the diversity of Pareto points offered when the exit point is added to the current set of exit points. The function takes as arguments: (1) the set of exit points that have been selected so far in constructing $\Gamma(N, k)$, (2) the set of candidate exit points to add to the set of selected exit points, and (3) the table of metric values for all of the exit points in $Y(N)$. The function *selectExit* returns an exit point candidate from the given set of candidate exit points (Argument 2) that maximizes Pareto point diversity when added to the set of previously-selected exit points (Argument 1).

Algorithm 3: An algorithm to construct the set of selected exits $\Gamma(N, k)$ for the elastic DCNN that is derived as part of the proposed design methodology.

input : Network N: DNN network that is to be elasticized.
input : k: number of desired network exits, $k \leq m'$.
input : Set $Y(N) = \{F_1, F_2, \ldots, F_{m'}\}$: collection of non-dominated early-exit candidates.
input : Table T: table containing metric values for all early-exit candidates in $Y(N)$.
output: Set *selectedExits*: set of selected early exits.
Procedure *Determine-exit-set* *N, k, Y(N), T, selectedExits*
    $selectedExits = \{F_1, F_{m'}\}$;
    for $i = 1 \rightarrow (k - 2)$ **do**
        $\gamma = Y(N) - selectedExits$;
        $new\_exit = select\_exit(selectedExits, \gamma, T)$;
        $selectedExits = selectedExits \cup \{new\_exit\}$;
    end
    return *selectedExits*
end

Our greedy exit selection algorithm together with the diversity-aware assessment

function *selectExit* provides an efficient means for selecting a diverse set of subnetworks of N that can be deployed in a compact manner. The efficiency of the selection process is important because the number of possible subnetworks of a given size k grows exponentially with the number of candidate exit points m in the network N. For state-of-the-art DCNNs, it is possible to have more than 50, and even hundreds of candidate exit points, so exhaustive evaluation of elastic network configurations becomes extremely time consuming or infeasible.

5.3 Experiment

In this section, we present experiments that demonstrate the utility of the proposed diversity-aware algorithm for configuring elastic DCNNs.

5.3.1 Dataset

We use the CIFAR-100 dataset[77]. CIFAR-100 contains 100 classes, with each class having 500 training samples and 100 testing samples. We refer to all of the 50 000 training samples across all 100 classes collectively as the CIFAR-100 training set. We split the CIFAR-100 training into two sets with 80% as our training set and 20% as a validation set. Our testing set uses the entire CIFAR-100 testing set, which consists of 10 000 images across all 100 classes.

5.3.2 Preprocessing and Training

Data augmentation is used in the experiments, including random crops with four paddings and a random horizontal flip. The ImageNet pre-trained weights are used to initialize the main backbone network, and other branches are initialized with random weights. All images are resized to size 224×224.

During the training phase, the initial learning rate is 10^{-3}. The loss function used for training consists of these five branches($\mathcal{L}_{100-classes}$, $\mathcal{L}_{20-classes}$, $\mathcal{L}_{10-classes}$, $\mathcal{L}_{5-classes}$, $\mathcal{L}_{mse}$). The losses from these five branches are summed up to form the total loss $\mathcal{L}$.

The learning rate is divided by 10 every time when the validation loss has not decreased for 8 epochs. During the inference phase, we use the output from the $\mathcal{L}_{100-classes}$ loss branch only as it turns out to be the most accurate predictor among all five branches and their combinations.

5.3.3 Evaluation Metrics

To evaluate the proposed approach, we experimented with three well-known DCNNs — DenseNet-121, Resnet-50, and EfficientNet-B0. We started our experimental setup for each network by deriving the corresponding elastic versions, which we refer to as Elastic-DenseNet-121, Elastic-ResNet-34, and Elastic-EfficientNet-B0. We derived these elastic versions by applying the elastic neural network design methodology presented in [78, 79] along with the training procedure described in Subsection 5.3.2.

The set of relevant implementation metrics defined for our experiments are the *error rate*, which is derived from the test accuracy of the three elastic neural networks, and

FLOPs, which stands for the number of Floating Point Operations to perform inference using the network up to a given exit point. For Elastic-DenseNet-121, 58 exits are created ($m = 58$), including 57 early exits and one exit at the end of the network. For Elastic-ResNet-50, we use $m = 16$ with 15 early exits and 1 exit appended to the ResNet-50 network backbone. Finally, for Elastic-EfficientNet-B0, we use $m = 7$ with 6 early exits.

Table 5.1, Table 5.2 and Table 5.3 provide detailed results for the two metrics of interest. These tables also provide data about whether or not each early exit is a Pareto point in relation to the design space created by all of the early exits.

Table 5.1: The FLOPs, accuracy, and error rate of Elastic-DenseNet-121 with the CIFAR-100 dataset.

Early Exit Index	**FLOPs**	**Accuracy (%)**	**Error Rate (%)**	**Pareto Point?**
1	8,205,912	20.9	79.1	✓
2	47,495,707	21.14	78.86	✓
3	81,276,231	22	78	✓
4	109,402,504	22.84	77.16	✓
5	156,347,284	22.11	77.89	✗
6	178,848,303	22.8	77.2	✗
7	205,089,826	25.91	74.09	✓
8	233,216,099	25.89	74.11	✗
9	263,227,122	28.73	71.27	✓
10	295,122,896	31.6	68.4	✓
11	328,903,420	33.57	66.43	✓

Continued on next page

Table 5.1 – Continued from previous page

Early Exit Index	FLOPs	Accuracy (%)	Error Rate (%)	Pareto Point?
12	364,568,695	35.42	64.58	✓
13	402,118,719	35.32	64.68	✗
14	441,553,494	39.2	60.8	✓
15	482,873,019	38.2	61.8	✗
16	526,077,295	42.04	57.96	✗
17	571,166,321	43.1	56.9	✗
18	647,948,147	45.22	54.78	✗
19	679,872,917	45.32	54.68	✗
20	516,682,540	44.4	55.6	✓
21	552,376,810	46.59	53.41	✓
22	589,955,831	48.63	51.37	✓
23	629,419,602	50.71	49.29	✓
24	670,768,123	51.8	48.2	✓
25	714,001,395	51.35	48.65	✗
26	759,119,417	55.1	44.9	✓
27	806,122,189	58.82	41.18	✓
28	853,153,957	55.12	44.88	✗
29	903,926,230	55.9	44.1	✗
30	956,583,253	59.14	40.86	✓

Continued on next page

Table 5.1 – Continued from previous page

Early Exit Index	FLOPs	Accuracy (%)	Error Rate (%)	Pareto Point?
31	1,011,125,026	61.15	38.85	✓
32	1,067,551,549	61.62	38.38	✓
33	1,125,862,823	60.17	39.83	✗
34	1,230,712,931	61.18	38.82	✗
35	1,294,649,459	59.1	40.9	✗
36	1,360,470,738	62.2	37.8	✓
37	1,428,176,767	62.21	37.79	✓
38	1,497,767,546	63.15	36.85	✓
39	1,569,243,075	63.2	36.8	✓
40	1,592,766,208	62.29	37.71	✗
41	1,668,011,239	65.66	34.34	✓
42	1,923,641,368	65.26	34.74	✗
43	1,970,644,140	63.27	36.73	✗
44	2,019,531,662	68.4	31.6	✓
45	2,070,303,935	67.25	32.75	✗
46	2,122,960,958	69.3	30.7	✓
47	2,177,502,731	68.5	31.5	✗
48	2,233,929,254	70.32	29.68	✓
49	2,292,240,528	69.38	30.62	✗

Continued on next page

Table 5.2: The FLOPs, accuracy, and error rate of Elastic-ResNet-50 with the CIFAR-100 dataset.

Early Exit Index	FLOPs	Accuracy (%)	Error Rate (%)	Pareto Point?
1	341,067,008	25.78	74.22	✓
2	551,729,664	26.77	73.23	✓
3	762,392,320	27.89	72.11	✓
4	1,057,880,320	30.76	69.24	✓
5	1,353,368,320	30.99	69.01	✓
6	1,571,785,984	37.09	62.91	✓
7	1,790,203,648	47.00	53.00	✓
8	2,085,639,936	46.89	53.11	✗
9	2,304,109,413	50.22	49.78	✓
10	2,522,578,789	52.02	47.98	✓
11	2,818,118,501	57.09	42.91	✓
12	3,036,587,877	69.34	30.66	✓
13	3,255,057,253	73.30	26.70	✓
14	3,550,700,389	75.33	24.67	✓
15	3,769,273,189	76.04	23.96	✓
16	3,987,845,989	78.04	21.96	✓

Table 5.1 – Continued from previous page

Early Exit Index	FLOPs	Accuracy (%)	Error Rate (%)	Pareto Point?
50	2,352,436,552	72.34	27.66	✓
51	2,414,517,326	75.27	24.73	✓
52	2,478,482,851	73.3	26.7	✗
53	2,544,333,126	73.37	26.63	✗
54	2,612,068,151	74.35	25.65	✗
55	2,679,832,172	74.39	25.61	✗
56	2,751,336,698	74.4	25.6	✗
57	2,824,725,974	75.61	24.39	✓
58	2,900,000,000	75.42	24.58	✓

Table 5.3: The FLOPs, accuracy, and error rate of Elastic-EfficientNet-B0 with the CIFAR-100 dataset.

Early Exit Index	FLOPs	Accuracy (%)	Error Rate (%)	Pareto Point?
1	561,341	12.39	87.61	✓
2	2,544,744	33.53	66.47	✓
3	4,528,148	47.39	52.61	✓
4	7,297,428	50.38	49.62	✓
5	10,066,708	53.12	46.88	✓
6	13,62,1865	70.22	29.78	✓
7	14,819,392	89.6	10.4	✓

5.3.4 Pareto Points and Diversity

With the selected metrics and Pareto points determined from Subsection 5.3.3, the dominated early exits and Pareto front can be readily derived from the data in Table 5.1, Table 5.2 and Table 5.3. The Pareto front and dominated design points for Elastic-DenseNet-121, Elastic-ResNet-50, and Elastic-EfficientNet-B0 are illustrated in Figure. 5.2, Figure. 5.3, and Figure. 5.4, respectively.

Once the Pareto points corresponding to the early exits are found, we can apply Algorithm 3 to extract the most diverse subset of early exits based on the desired number of exits k. The value of k might be chosen, for example, based on the overall network size that can be stored efficiently within the resource constraints of the targeted processing platform. Recall that each early exit requires a separate classification layer (early-exit layer); these layers may require significant numbers of parameters to ensure sufficient accuracy.

The DCI index, which is used in Algorithm 3, involves a parameter called `div`, which is used to divide the given multi-dimensional design evaluation space into a grid; the value of `div` determines the size (granularity) of the cells in the grid. Careful setting of

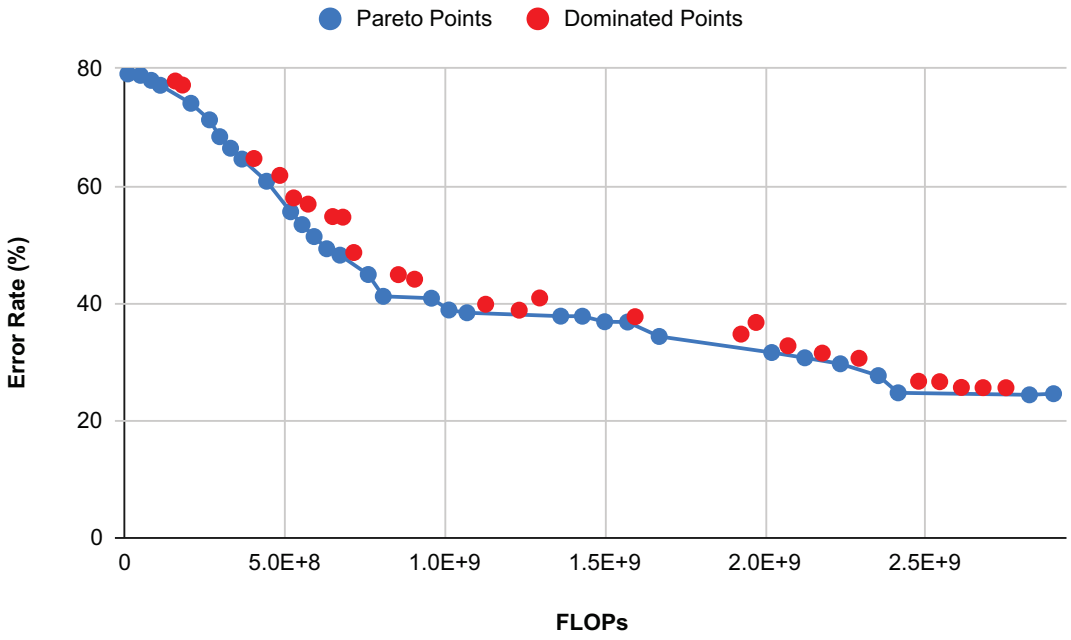


Figure 5.2: Pareto front and dominated design points of the Elastic-DenseNet-121 network with the CIFAR-100 dataset.

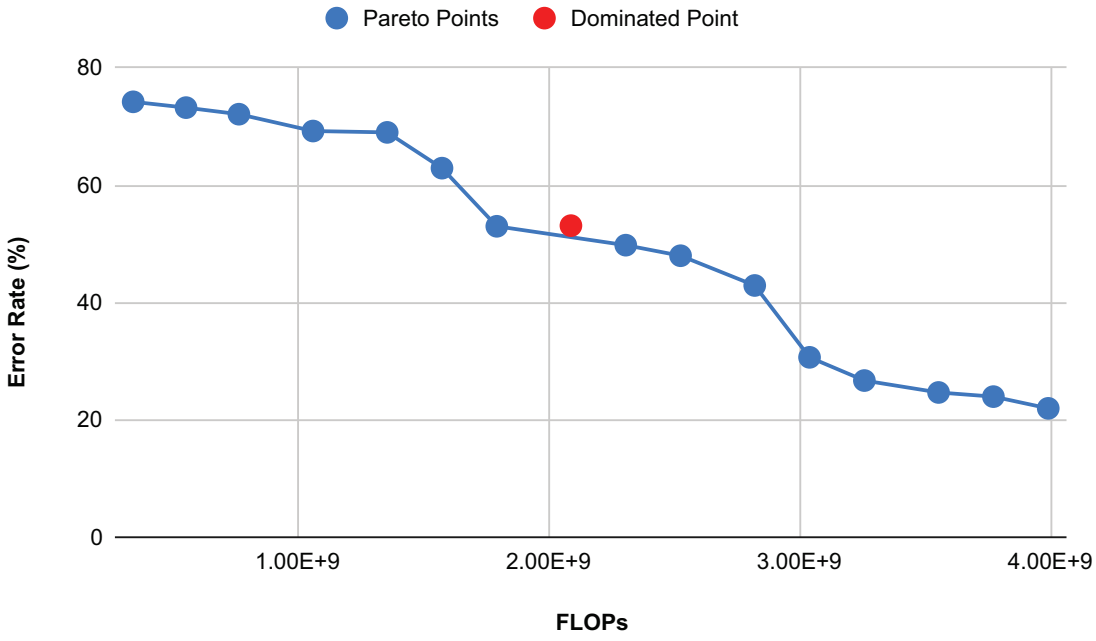


Figure 5.3: Pareto front and dominated design points of the Elastic-ResNet-50 network with the CIFAR-100 dataset.

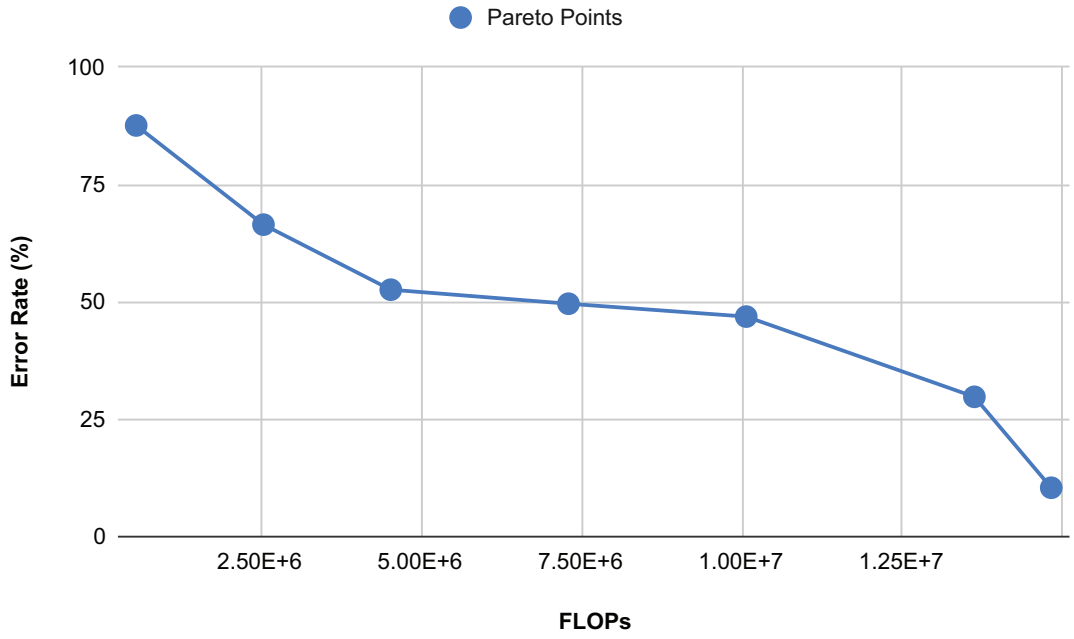


Figure 5.4: Pareto front and dominated design points of the Elastic-EfficientNet-B0 network with the CIFAR-100 dataset.

`div` is important to ensure derivation of design point subsets that have high diversity [86]. In our experiments, we apply methods for optimal setting of `div` that are presented in [86]. The methods rely on characteristics of the overall set of design points ($Y(N)$ in our case) from which we are extracting a compact, Pareto-optimal subset. The values of `div` that are derived in this way for our experiments are 11, 8, and 5, respectively, for Elastic-DenseNet-121, Elastic-ResNet-50, and Elastic-EfficientNet-B0.

Table 5.4, Table 5.6, and Table 5.5 summarize the outcomes from applying Algorithm 3 to the complete set of Pareto points $Y(N)$ for each of the three networks, Elastic-DenseNet-121, Elastic-ResNet-50, and Elastic-EfficientNet-B0. These results help to validate and concretely demonstrate the proposed methodology for deriving compact, adaptation-friendly neural networks that can operate over diverse operational characteris-

Table 5.4: Early exit subset selections with different slection subset size k for Eelastic-DenseNet-121.

Desired Number of Early Exits k	**Early Exit Index/Indices**
1	23
2	21, 23
3	21, 23, 39
4	11, 21, 23, 39
5	11, 21, 23, 39, 44
6	11, 21, 23, 39, 44
7	11, 21, 23, 27, 39, 44
8	11, 21, 23, 27, 36, 39, 44
9	11, 21, 22, 23, 27, 36, 39, 44
10	11, 21, 22, 23, 27, 36, 39, 44, 48
11	11, 21, 22, 23, 27, 36, 39, 44, 48
12	11, 12, 21, 22, 23, 27, 36, 39, 44, 48
13	11, 12, 21, 22, 23, 27, 36, 39, 44, 48
14	11, 12, 21, 22, 23, 27, 36, 39, 41, 44, 48
15	11, 12, 21, 22, 23, 27, 31, 36, 39, 41, 44, 48
16	11, 12, 21, 22, 23, 27, 30, 31, 36, 39, 41, 44, 48
17	11, 12, 20, 21, 22, 23, 27, 30, 31, 36, 39, 41, 44, 48
18	11, 12, 20, 21, 22, 23, 24, 27, 30, 31, 36, 39, 41, 44, 48
19	11, 12, 20, 21, 22, 23, 24, 27, 30, 31, 36, 39, 41, 44, 48, 51
20	9, 11, 12, 20, 21, 22, 23, 24, 27, 30, 31, 36, 39, 41, 44, 48, 51
21	9, 11, 12, 20, 21, 22, 23, 24, 27, 30, 31, 36, 39, 41, 44, 48, 51
22	7, 9, 11, 12, 20, 21, 22, 23, 24, 27, 30, 31, 36, 39, 41, 44, 48, 51
23	3, 7, 9, 11, 12, 20, 21, 22, 23, 24, 27, 30, 31, 36, 39, 41, 44, 48, 51
24	3, 4, 7, 9, 11, 12, 20, 21, 22, 23, 24, 27, 30, 31, 36, 39, 41, 44, 48, 51
25	3, 4, 7, 9, 11, 12, 15, 20, 21, 22, 23, 24, 27, 30, 31, 36, 39, 41, 44, 48, 51
26	3, 4, 7, 9, 11, 12, 15, 20, 21, 22, 23, 24, 27, 30, 31, 36, 38, 39, 41, 44, 48, 51
27	3, 4, 7, 9, 11, 12, 15, 20, 21, 22, 23, 24, 27, 30, 31, 32, 36, 38, 39, 41, 44, 48, 51
28	2, 3, 4, 7, 9, 11, 12, 15, 20, 21, 22, 23, 24, 27, 30, 31, 32, 36, 38, 39, 41, 44, 48, 51
29	2, 3, 4, 7, 9, 11, 12, 14, 15, 20, 21, 22, 23, 24, 27, 30, 31, 32, 36, 38, 39, 41, 44, 48, 51
30	2, 3, 4, 7, 9, 11, 12, 14, 15, 20, 21, 22, 23, 24, 27, 30, 31, 32, 36, 38, 39, 41, 44, 48, 50, 51
31	2, 3, 4, 7, 9, 11, 12, 14, 15, 20, 21, 22, 23, 24, 27, 30, 31, 32, 36, 38, 39, 41, 44, 46, 48, 50, 51

Table 5.5: Early exit subset selections with different selection subset size k for Elastic-EfficientNet-B0.

Desired Number of Early Exits k	**Early Exit Index/Indices**
1	3
2	3, 4
3	3, 4, 5
4	2, 3, 4, 5
5	2, 3, 4, 5, 6

Table 5.6: Early exit subset selections with different selection subset size k for Elastic-ResNet-50.

Desired Number of Early Exits k	**Early Exit Index/Indices**
1	9
2	9, 10
3	5, 9, 10
4	5, 9, 10, 12
5	5, 7, 9, 10, 12
6	5, 7, 9, 10, 11, 12
7	4, 5, 7, 9, 10, 11, 12
8	4, 5, 6, 7, 9, 10, 11, 12
9	4, 5, 6, 7, 9, 10, 11, 12, 14
10	2, 4, 5, 6, 7, 9, 10, 11, 12, 14
11	2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 14
12	2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 14, 15
13	2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15

tics.

5.4 Additional Results

In this section, we present additional experimental results to show how the proposed multi-objective CNN implementation framework can be generalized to a wider range of tasks.

The core aspects of the approach laid out in Chapter 5 are not only applicable to elastic neural networks; the core approach can be applied for adaptive system design for other types of complex embedded knowledge extraction systems. In this section, we demonstrate this more general applicability using a (non-elastic) DCNN-based face identification model. In our experiments, we employ a dataset consisting of 574 face images with females and males taken by a Nikon D5600 camera. The experiment is executed on

an x86 desktop with an Intel(R) Core(TM) i7-2600K CPU@3.40GHz, NVIDIA GeForce GTX 1080, and 16GB RAM.

We have experimented with various alternative implementation configurations for the face identification model, and applied our proposed multi-objective configuration selection algorithm to extract strategic design options for deployment. We have made measurements of different implementation configurations running on both CPU and GPU devices, and applied our selection algorithm using data derived from these measurements.

The CPU-targeted and GPU-targeted system configurations and associated measurements are presented in Table 5.7 and Table 5.8. Pareto points in Table 5.7 and Table 5.8 are marked with ✓. The Pareto-optimal configurations are indexed from 1, from the top row of the table to the bottom row. In total, 10 Pareto-optimal configurations are derived from the CPU-targeted results from among all of the configurations evaluated. Similarly, a total of 3 Pareto-optimal configurations are derived from the GPU-targeted results.

Table 5.9 summarizes the selection process across the CPU-targeted candidate configurations using our developed selection algorithm. Table 5.9 also summarizes the selection results for different desired numbers of CPU-targeted configurations. Similarly, Table 5.10 illustrates the selection process and selection results for GPU-targeted configurations.

A minor difference between the selection algorithm used here compared to Algorithm 3 is that in our experiments with the face identification model, no configurations are assumed to be in the selection subset from the beginning.

Having multiple design options for the face identification model is useful if the

Table 5.7: CPU-targeted system configurations and their associated performance metrics.

Pipeline Stages	Intra-op Threads	Inference Latency (s)	Peak Memory Consumption (MB)	Average Power Consumption (W)	Pareto Point?
1	1	14.72	154.08	34.79	✓
	2	10.45	154.02	51.21	✓
	3	9.31	154.21	64.42	✓
	4	9.39	153.89	73.79	✓
	5	10.31	154.12	75.03	✗
	6	9.91	154.13	77.95	✗
	7	9.94	154.57	84.73	✗
	8	9.89	155.21	89.72	✗
2	1	8.26	158.52	57.71	✓
	2	8.38	164.72	74.80	✗
	3	8.38	161.75	88.38	✗
	4	8.62	160.15	90.82	✗
3	1	6.97	167.94	71.35	✓
	2	6.71	163.30	95.24	✓
4	1	6.10	175.34	82.26	✓
	2	6.79	172.10	105.37	✗
5	1	5.79	188.32	96.64	✓
6	2	5.66	176.16	99.18	✓
7	3	5.73	195.62	108.81	✗
8	4	6.65	174.77	111.83	✓

Table 5.8: GPU-targeted system configurations and their associated performance metrics.

Pipeline Stages	Inference Latency (s)	Peak RAM Consumption (MB)	Peak GPU Memory Consumption (MB)	Average CPU + GPU Power Consumption (W)	Pareto Point?
1	4.99	3539.49	875	75.17	✓
2	3.57	3548.03	891	94.73	✓
3	3.31	3550.70	901	101.67	✓
4	3.59	3549.44	911	109.29	✗
5	3.54	3554.79	924	112.85	✗
6	3.58	3553.55	932	117.73	✗
7	3.94	3553.97	932	122.11	✗
8	4.63	3553.11	942	129.69	✗

same dataflow functionality is being deployed in different applications (different deployments). There might, for example, be two versions of a product that have different price points (low-end and high-end) or that have different energy constraints or processing constraints depending on how lean the target platform is. In such cases, there may just be one configuration per deployment. However, the overall product suite may involve multiple configurations, and it is useful in such a context to derive the selected set of configurations from a diverse space of optimized candidate solutions.

5.5 Conclusion

In this chapter, we have developed a framework for efficiently selecting sets of early exit points for elastic neural networks. The framework emphasizes the derivation of exit

Table 5.9: Selected CPU-targeted configurations with the proposed algorithm for the face identification model.

Desired Number of Configurations k	**Configuration Index/Indices**
1	1
2	1, 7
3	1, 7, 8
4	1, 7, 8, 4
5	1, 7, 8, 4, 5
6	1, 7, 8, 4, 5, 2
7	1, 7, 8, 4, 5, 2, 6
8	1, 7, 8, 4, 5, 2, 6, 10
9	1, 7, 8, 4, 5, 2, 6, 10, 9
10	1, 7, 8, 4, 5, 2, 6, 10, 9, 3

Table 5.10: Selected GPU-targeted configurations with the proposed algorithm for the face identification model

Desired Number of Configurations k	**Configuration Index/Indices**
1	2
2	1, 2
3	1, 2, 3

points in the network that collectively provide diversity in the trade-offs that are delivered by the elastic network. The diversity of trade-offs is addressed objectively using a concept from multiobjective optimization called the diversity comparison indicator (DCI). The effectiveness of the proposed approach was demonstrated using popular deep convolutional neural networks. In contrast with conventional methods for runtime adaptiveness, such as those that rely on ensemble models, the approach proposed in this chapter involves elastic neural network design integrated with a DCI assessment metric and an efficient process for selecting early exit points to elasticize the given neural network. This integration improves the adaptivity and compactness of the derived DCNNs. Interesting directions for future work include extending the framework to handle heterogeneous platforms, where performance metrics may vary depending on how the network is partitioned across different hardware subsystems.

Chapter 6

Conclusions and Future Work

In this chapter, we summarize the contributions made in this thesis and explore potential directions for future research and the associated challenges within these areas.

6.1 Conclusions

In this thesis, we have addressed four topics in the efficient simulation and implementation of neural networks on resource-constrained platforms. We have also presented our research contributions in these topics:

- An efficient methodology and software tool for dataflow-based simulation of spiking neural networks.
- A novel architecture for resource-constrained implementation of neural networks for hyperspectral image classification.
- A software package for optimized edge-based DCNN inference using a lightweight-dataflow-based design methodology.
- A framework for configuring DCNNs with early-exit points and efficiently selecting a strategic subset of early-exit points based on multiobjective considerations.

These contributions provide systematic approaches for tackling neural network simulation and implementation challenges on resource-constrained platforms, from high-

level neural network architectural design for neuromorphic systems, to adaptive machine learning systems with CNNs, to resource-constrained systems that involve complex CNN task scheduling and optimization constraints. Our contributions also address challenges imposed by applications involving real-time hyperspectral image classification with enriched spectral dimension information.

6.2 Future Work

6.2.1 Simulation of Spiking Neural Networks

There are several useful directions to explore in the field of SNN simulation, including methods for handling a wider variety of neuron models and providing improved simulation efficiency.

First, we discuss the problem of incorporating various spiking neuron models into SNN simulations. For instance, the Integrate-and-Fire neuron model has demonstrated both effectiveness and computational efficiency, allowing neural networks built with Integrate-and-Fire neurons to achieve comparable performance to their CNN counterparts in object detection tasks [90] and image classification [91]. These findings highlight the potential of spiking neuron models and spiking neural networks to enhance the efficiency of state-of-the-art computer vision systems based on CNNs, particularly due to their superior energy efficiency when deployed on specialized neuromorphic hardware, such as IBM TrueNorth [92] or Intel Loihi [93], which are specifically designed for SNN simulations.

The application of alternative neuron models — beyond the Integrate-and-Fire model — to computer vision is an interesting area for further work. Such models need to be con-

sidered in terms of their impact on the accuracy of the targeted image analysis tasks as well as their potential for energy-efficient hardware implementation. We envision that the SNN simulation framework proposed in this thesis can be of great utility in exploring the application of alternative neuron models for the design and implementation of neuromorphic computer vision systems.

A second interesting direction for future work on SNN simulation is exploring further improvements in simulation efficiency. Event-driven simulation has already been established as more efficient than time-driven approaches. Nevertheless, the computational complexity associated with conducting Finite Element Analysis (FEA) remains high for large-scale SNNs. Therefore, neuron updates via lookup tables or by utilizing multi-threaded or GPU-accelerated simulation techniques can significantly enhance simulation efficiency. Using a pre-computed lookup table for neuron status, considering different input-output pairs, can reduce the computation required at runtime, at the expense of frequent memory access during the simulation. The trade-off between lookup-table- and FEA-based approaches can vary significantly depending on the simulation platform, and streamlined selection of the lookup table size can improve the overall simulation latency at the cost of precision.

Employing multi-threaded or GPU-accelerated simulation techniques can greatly enhance efficiency by concurrently updating the states of multiple neurons at a given time step, as demonstrated in [94]. While most studies on parallel implementation focus on exploiting parallelization for neuron updates in time-driven simulation, event-driven simulation also holds promise for the application of parallelization techniques.

6.2.2 Real-Time Hyperspectral Image Classification

Carrying out real-time hyperspectral image classification has always been a challenging task. To facilitate research in this area, two potential directions can be explored: systematic approaches for HSI dimension reduction and inference acceleration.

HSI dimension reduction focuses on reducing the spectral complexity of an HSI before providing it as input to a classifier. Existing approaches include:

- Principle Component Analysis (PCA): This technique transforms the input HSI in an effort to concentrate most of the information within a small subset of dimensions; the transformed, dimension-reduced version of the input signal is then used for classification.

- Band subset selection: This approach involves selecting a subset of spectral bands from the available bands based on a given selection rule. Although the selection rule is typically pre-defined, it is also possible to apply dynamically-varying selection rules.

Both approaches, PCA and band subset selection, have proven to be effective in reducing the computational complexity of HSI processing tasks [95, 39]. Applying the discrete cosine transform (DCT) [96] to HSIs holds promise for further improvement.

Inference acceleration using GPUs or accelerators can significantly improve inference speed, which is important for real-time HSI analysis. While applying accelerators to deep neural networks (DNNs) has become a standard approach for both training and inference in many applications, their application to HSI problems such as VBIC has not

been extensively explored. This direction shows promise in significantly enhancing the real-time performance of HSI classification systems.

6.2.3 Multi-Objective Design Optimization for DNN Inference

In this section, we discuss three interesting directions for future work in multi-objective design optimization for DNN inference.

The first direction is extending our multi-objective exit-point configuration framework to handle heterogeneous platforms. This involves considering performance metrics that may vary depending on how the network is partitioned across different hardware subsystems. By accommodating heterogeneous platforms, the framework can provide more flexibility and optimize DNN inference for a broader range of hardware configurations.

The second direction involves the extension of LCIP beyond C/C++. Currently, LCIP supports implementations based on programming languages like C/C++. Exploring the inclusion of hardware description languages (HDLs) can expand the scope of LCIP and enable the optimization of DNN inference on hardware platforms that utilize HDL-based design flows. Such an extension of LCIP can enhance the compatibility, applicability, and customizability of LCIP in diverse hardware environments.

The third direction is integrating LCIP with additional methods for automated dataflow graph partitioning and scheduling. By integrating LCIP with other automated techniques for dataflow graph partitioning and scheduling, the framework can further optimize the allocation of computational resources, task parallelism, and dataflow efficiency. This integration can improve the overall performance and resource utilization of DNN inference

on resource-constrained platforms.

These future research directions can enhance the capabilities and versatility of our developed multi-objective design optimization framework for DNN inference, enabling its application in a wider range of scenarios and facilitating efficient utilization of diverse hardware platforms.

Bibliography

[1] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, vol. 4, no. 11, 2018.

[2] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, 2018.

[3] C. J. Wu et al., "Machine learning at facebook: Understanding inference at the edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344.

[4] 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.

[5] Lei Pan, Francois Christophe, Tommi Mikkonen, Zhu Li, and Shuvra S Bhattacharyya, "Simulating spiking neural networks with timed dataflow graphs," in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 2020, pp. 64–68.

[6] E. Painkras et al., "SpiNNaker: A 1-W 18-core system-on-chip for massively-parallel neural network simulation," *IEEE Journal of Solid State Circuits*, vol. 48, no. 8, pp. 1943–1953, 2013.

[7] F. Akopyan et al., "TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015.

[8] C. Shen et al., "A lightweight dataflow approach for design and implementation of SDR systems," in *Proceedings of the Wireless Innovation Conference and Product Exposition*, 2010, pp. 640–645.

[9] M. L. Hines and N. T. Carnevale, "NEURON: A Tool for Neuroscientists," 2001.

[10] J. M. Eppler, "PyNEST: A Convenient Interface to the NEST Simulator," *Frontiers in Neuroinformatics*, 2008.

[11] D. Goodman and R. Brette, "Brian: A simulator for spiking neural networks in Python," *Frontiers in Neuroinformatics*, vol. 2, no. 5, pp. 1–10, 2008.

[12] M. Stimberg, D. F. M. Goodman, V. Benichoux, and R. Brette, "Equation-oriented Specification of Neural Models for Simulations," *Frontiers in Neuroinformatics*, 2014.

[13] F. Naveros, J. A. Garrido, R. R. Carrillo, E. Ros, and N. R. Luque, "Event- and time-driven techniques using parallel CPU-GPU co-processing for spiking neural networks," *Frontiers in Neuroinformatics*, vol. 11, 2017, Article 7.

[14] K. Cheung, S. R. Schultz, and W. Luk, "NeuroFlow: A General Purpose Spiking Neural Network Simulation Platform using Customizable Processors," *Frontiers in Neuroscience*, vol. 9, no. JAN, 2016.

[15] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[16] R. Brette and W. Gerstner, "Adaptive exponential integrate-and-fire model as an effective description of neuronal activity," *Journal of Neurophysiology*, vol. 94, no. 5, pp. 3637–3642, 2005, PMID: 16014787.

[17] C. Koch and I. Segev, *Methods in Neuronal Modeling: From Ions to Networks*, MIT press, 1998.

[18] G. Bi and M. Poo, "Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type," *Journal of Neuroscience*, vol. 18, no. 24, pp. 10464–10472, 1998.

[19] L. Li, P. Deaville, A. Sapio, L. Anttila, M. E. Valkama, M. Wolf, and S. Bhattacharyya, "A framework for design and implementation of adaptive digital predistortion systems," in *Proceedings of the IEEE International Conference on Artificial Intelligence Circuits and Systems*, Hsinchu, Taiwan, March 2019, pp. 112–116.

[20] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, pp. 773–799, May 1995.

[21] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, Springer, third edition, 2019.

[22] J. Geng, H. Li, Y. Liu, Y. Liu, M. Kashef, R. Candell, and S. S. Bhattacharyya, "Model-based cosimulation for industrial wireless networks," in *Proceedings of the IEEE International Workshop on Factory Communication Systems*, 2018, pp. 1–10.

[23] L. Pan, R. Liao, Z. Li, and S. S. Bhattacharyya, "Dynamic, data-driven hyperspectral image classification on resource-constrained platforms," in *Dynamic Data Driven Applications Systems: Third International Conference, DDDAS 2020, Boston, MA, USA, October 2-4, 2020, Proceedings 3*. Springer, 2020, pp. 320–327.

[24] P. WT. Yuen and M. Richardson, "An introduction to hyperspectral imaging and its application for security, surveillance and target acquisition," *The Imaging Science Journal*, vol. 58, no. 5, pp. 241–253, 2010.

[25] J. Freeman et al., "Multispectral and hyperspectral imaging: Applications for medical and surgical diagnostics," in *Proceedings of the International Conference of the IEEE Engineering in Medicine and Biology Society*, 1997, pp. 700–701.

[26] G.J. Edelman, E.Gaston, T.G. van Leeuwen, P.J. Cullen, and M.C.G. Aalders, "Hyperspectral imaging for non-contact analysis of forensic traces," *Forensic science international*, vol. 223, no. 1-3, pp. 28–39, 2012.

[27] S. B. Serpico and L. Bruzzone, "A new search algorithm for feature selection in hyperspectral remote sensing images," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 39, no. 7, pp. 1360–1367, 2001.

[28] J. Patrick, R. Brant, and E. Blasch, "Hyperspectral imagery throughput and fusion evaluation over compression and interpolation," in *Proceedings of the International Conference on Information Fusion*, 2008, pp. 1–8.

[29] S. Li, W. Song, L. Fang, Y. Chen, P. Ghamisi, and J. A. Benediktsson, "Deep learning for hyperspectral image classification: An overview," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 57, no. 9, pp. 6690–6709, 2019.

[30] D. Madroñal, R. Lazcano, R. Salvador, H. Fabelo, S. Ortega, G. M. Callico, E. Juarez, and C. Sanz, "SVM-based real-time hyperspectral image classifier on a manycore architecture," *Journal of Systems Architecture*, vol. 80, pp. 30–40, 2017.

[31] S. M. Chai, A. Gentile, W. E. Lugo-Beauchamp, J. Fonseca, J. L. Cruz-Rivera, and D. S. Wills, "Focal-plane processing architectures for real-time hyperspectral image processing," *Applied Optics*, vol. 39, no. 5, pp. 835–849, 2000.

[32] Y. Luo, J. Zou, C. Yao, X. Zhao, T. Li, and G. Bai, "HSI-CNN: A novel convolution neural network for hyperspectral image," in *Proceedings of the International Conference on Audio, Language and Image Processing*, 2018, pp. 464–469.

[33] A. B. Hamida, A. Benoit, P. Lambert, and C. B. Amar, "3-D deep learning approach for remote sensing image classification," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 56, no. 8, pp. 4420–4434, 2018.

[34] Y. Li, H. Zhang, and Q. Shen, "Spectral–spatial classification of hyperspectral imagery with 3D convolutional neural network," *Remote Sensing*, vol. 9, no. 1, pp. 1–21, 2017.

[35] M. He, B. Li, and H. Chen, "Multi-scale 3D deep convolutional neural network for hyperspectral image classification," in *Proceedings of the International Conference on Image Processing*, 2017, pp. 3904–3908.

[36] Z. Wu, Q. Wang, A. Plaza, J. Li, L. Sun, and Z. Wei, "Real-time implementation of the sparse multinomial logistic regression for hyperspectral image classification on GPUs," *IEEE Geoscience and Remote Sensing Letters*, vol. 12, no. 7, pp. 1456–1460, 2015.

[37] V. Sharma, A. Diba, T. Tuytelaars, and L. Van Gool, "Hyperspectral cnn for image classification & band selection, with application to face recognition," Tech. Rep. KUL/ESAT/PSI/1604, KU Leuven, 2016.

[38] H. Li, Y. Liu, K. Sudusinghe, J. Yoon, E. Blasch, M. van der Schaar, and S. S. Bhattacharyya, "Design of a dynamic data-driven system for multispectral video processing," in *Handbook of Dynamic Data Driven Applications Systems*, pp. 529–545. Springer, 2018.

[39] H. Li, L. Pan, E. J. Lee, Z. Li, M. J. Hoffman, A. Vodacek, and S. S. Bhattacharyya, "Hyperspectral video processing on resource-constrained platforms," in *Proceedings of the Workshop on Hyperspectral Image and Signal Processing*, 2019.

[40] H. Li, K. Sudusinghe, Y. Liu, J. Yoon, M. van der Schaar, E. Blasch, and S. S. Bhattacharyya, "Dynamic, data-driven processing of multispectral video streams," *IEEE Aerospace & Electronic Systems Magazine*, vol. 32, no. 7, pp. 50–57, 2017.

[41] S. Lin, Y. Liu, K. Lee, L. Li, W. Plishker, and S. S. Bhattacharyya, "The DSPCAD framework for modeling and synthesis of signal processing systems," in *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich, Eds., pp. 1–35. Springer, 2017.

[42] M. F. Baumgardner, L. L. Biehl, and D. A. Landgrebe, "220 band AVIRIS hyperspectral image data set: June 12, 1992 Indian Pine Test Site 3," 2015.

[43] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization.," *Journal of machine learning research*, vol. 12, no. 7, pp. 2121–2159, 2011.

[44] N. Audebert, Bertrand Le Saux, and S. Lefevre, "Deep learning for classification of hyperspectral data: A comparative review," *IEEE Geoscience and Remote Sensing Magazine*, vol. 7, no. 2, pp. 159–173, 2019.

[45] M. Fauvel, J. Benediktsson, J. Chanussot, and J. Sveinsson, "Spectral and spatial classification of hyperspectral data using svms and morphological profiles," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 46, no. 11, pp. 3804–3814, 2008.

[46] A. Neuenschwander, M. Crawford, and S. Ringrose, "Results from the eo-1 experiment—a comparative study of earth observing-1 advanced land imager (ali) and landsat etm+ data for land cover mapping in the okavango delta, botswana," *International Journal of Remote Sensing*, vol. 26, no. 19, pp. 4321–4337, 2005.

[47] G. Naik, *Advances in Principal Component Analysis: Research and Development*, Springer, 2017.

[48] L. Pan, Y. Zhang, and S. S. Bhattacharyya, "LCIP: a retargetable framework for optimized cnn inference," in *Real-Time Image Processing and Deep Learning 2023*. SPIE, 2023, vol. 12528, pp. 17–32.

[49] T. Chen et al., "TVM: An automated end-to-end optimizing compiler for deep learning," in *USENIX Symposium on Operating Systems Design and Implementation*, Carlsbad, CA, 2018, pp. 579–594, USENIX Association.

[50] S. Minakova, E. Tang, and T. Stefanov, "Combining task- and data-level parallelism for high-throughput CNN inference on embedded CPUs-GPUs MPSoCs," in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Samos, Greece, 2020, pp. 18–35, Springer.

[51] S. Lin, J. Wu, and S. S. Bhattacharyya, "Memory-constrained vectorization and scheduling of dataflow graphs for hybrid CPU-GPU platforms," *ACM Transactions on Embedded Computing Systems*, vol. 17, no. 2, pp. 50:1–50:25, January 2018.

[52] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," Tech. Rep., Google Research, November 2015, Preliminary White Paper.

[53] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," 2019, arXiv:1912.01703v1 [cs.LG].

[54] M. Dukhan, Y. Wu, and H. Lu, "Qnnpack: Open source library for optimized mobile deep learning," `https://github.com/pytorch/QNNPACK`, 2018.

[55] C. Hu and B. Li, "Distributed inference with deep learning models across heterogeneous edge devices," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, London, UK, 2022, pp. 330–339, IEEE.

[56] B. Li, V. Gadepally, S. Samsi, M. Veillette, and D. Tiwari, "Serving machine learning inference using heterogeneous hardware," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, Boston, MA, USA, 2021, pp. 1–8, IEEE.

[57] R. Xie, H. Huttunen, S. Lin, S. S. Bhattacharyya, and J. Takala, "Resource-constrained implementation and optimization of a deep neural network for vehicle classification," in *2016 24th European Signal Processing Conference (EUSIPCO)*, Budapest, Hungary, 2016, pp. 1862–1866, IEEE.

[58] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[59] D. Luenemann, M. Fakih, and K. Gruettner, "Capturing neural-networks as synchronous dataflow graphs," in *MBMV 2020-Methods and Description Languages for Modelling and Verification of Circuits and Systems; GMM/ITG/GI-Workshop*, Stuttgart, Germany, 2020, pp. 1–10, VDE.

[60] S. S. Veeramanikandan et al., "Data flow and distributed deep neural network based low latency IoT-edge computation model for big data environment," *Engineering Applications of Artificial Intelligence*, vol. 94, pp. 1–8, 2020.

[61] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: Mapping regular and irregular convolutional neural networks on FPGAs," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 2, pp. 326–342, 2019.

[62] S. Song et al., "DFSynthesizer: Dataflow-based synthesis of spiking neural networks to neuromorphic hardware," 2021, arXiv:2108.02023 [cs.NE].

[63] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*, Stockholm, Sweden, 1974, pp. 471–475, IFIP.

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

[65] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, Salt Lake City, UT, USA, 2018, pp. 4510–4520, IEEE.

[66] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[67] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F.-F. Li, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, Miami, FL, USA, 2009, pp. 248–255, IEEE.

[68] C. Ledig, L. Theis, F. Huszár, J. Caballero, et al., "Photo-realistic single image super-resolution using a generative adversarial network," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4681–4690.

[69] J. Johnson, A. Alahi, and F-F. Li, "Perceptual losses for real-time style transfer and super-resolution," in *European conference on computer vision*. Springer, 2016, pp. 694–711.

[70] C. Dong, C. C. Loy, K. He, and X. Tang, "Image super-resolution using deep convolutional networks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 2, pp. 295–307, 2015.

[71] J. Kim, J. K. Lee, and K. M. Lee, "Accurate image super-resolution using very deep convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 1646–1654.

[72] B. Lim, S. Son, H. Kim, S. Nah, and K. M. Lee, "Enhanced deep residual networks for single image super-resolution," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2017, pp. 136–144.

[73] V. Sagar, "A pytorch tutorial to super-resolution," `https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Super-Resolution`, 2020.

[74] L. Du, A. Ho, and R. Cong, "Perceptual hashing for image authentication: A survey," *Signal Processing: Image Communication*, vol. 81, pp. 115713, 2020.

[75] L. Pan, Y. Zhang, Y. Zhou, H. Huttunen, and S. S. Bhattacharyya, "Multi-objective design optimization for image classification using elastic neural networks," in *2023 57th Annual Conference on Information Sciences and Systems (CISS)*. IEEE, 2023, pp. 1–6.

[76] O. Russakovsky et al., "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, pp. 211–252, 2015.

[77] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[78] Y. Bai, S. S. Bhattacharyya, A. P. Happonen, and H. Huttunen, "Elastic neural networks: A scalable framework for embedded computer vision," in *Proceedings of the European Signal Processing Conference*, Rome, Italy, September 2018, pp. 1472–1476.

[79] Y. Zhou, Y. Bai, S. S. Bhattacharyya, and H. Huttunen, "Elastic neural networks for classification," in *Proceedings of the IEEE International Conference on Artificial Intelligence Circuits and Systems*, Hsinchu, Taiwan, March 2019, pp. 251–255.

[80] Y. Jin and B. Sendhoff, "Pareto-based multiobjective machine learning: An overview and case studies," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, no. 3, pp. 397–415, 2008.

[81] E. J. Chen and L. H. Lee, "A multi-objective selection procedure of determining a pareto set," *Computers & Operations Research*, vol. 36, no. 6, pp. 1872–1879, 2009.

[82] M. Loni, S. Sinaei, A. Zoljodi, M. Daneshtalab, and M. Sjödin, "Deepmaker: A multi-objective optimization framework for deep neural networks in embedded systems," *Microprocessors and Microsystems*, vol. 73, pp. 1–11, 2020.

[83] A. Freitas, "A critical review of multi-objective optimization in data mining: a position paper," *ACM SIGKDD Explorations Newsletter*, vol. 6, no. 2, pp. 77–86, 2004.

[84] A. Teerapittayanon, B. McDanel, and H-T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd International Conference on Pattern Recognition (ICPR)*. IEEE, 2016, pp. 2464–2469.

[85] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Weinberger, "Multi-scale dense networks for resource efficient image classification," in *International Conference on Learning Representations*, Vancouver, Canada, April 2018.

[86] M. Li, S. Yang, and X. Liu, "Diversity comparison of pareto front approximations in many-objective optimization," *IEEE Transactions on Cybernetics*, vol. 44, no. 12, pp. 2568–2584, 2014.

[87] S. Mostaghim and J. Teich, "A New Approach on Many Objective Diversity Measurement," in *Practical Approaches to Multi-Objective Optimization*, Jürgen Branke, Kalyanmoy Deb, Kaisa Miettinen, and Ralph E. Steuer, Eds. 2005, vol. 4461 of *Dagstuhl Seminar Proceedings (DagSemProc)*, pp. 1–15, Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[88] O. Schutze, X. Esquivel, A. Lara, and C. Coello, "Using the averaged hausdorff distance as a performance measure in evolutionary multiobjective optimization," *IEEE Transactions on Evolutionary Computation*, vol. 16, no. 4, pp. 504–522, 2012.

[89] G. Lizarraga-Lizarraga, A. Hernandez-Aguirre, and S. Botello-Rionda, "G-metric: an m-ary quality indicator for the evaluation of non-dominated sets," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, 2008, pp. 665–672.

[90] S. Kim, S. Park, B. Na, and S. Yoon, "Spiking-yolo: Spiking neural network for energy-efficient object detection," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 07, pp. 11270–11277, Apr. 2020.

[91] B. Rueckauer, I-A. Lungu, Y. Hu, M. Pfeiffer, and S-C. Liu, "Conversion of continuous-valued deep networks to efficient event-driven networks for image classification," *Frontiers in neuroscience*, vol. 11, pp. 682, 2017.

[92] M. DeBole, B. Taba, A. Amir, F. Akopyan, et al., "Truenorth: Accelerating from zero to 64 million neurons in 10 years," *Computer*, vol. 52, no. 5, pp. 20–29, 2019.

[93] C-W. Lin, A. Wild, G. Chinya, Y. Cao, et al., "Programming spiking neural networks on intel's loihi," *Computer*, vol. 51, no. 3, pp. 52–61, 2018.

[94] D. Alevi, M. Stimberg, H. Sprekeler, K. Obermayer, and M. Augustin, "Brian2cuda: flexible and efficient simulation of spiking neural network models on gpus," *Frontiers in Neuroinformatics*, vol. 16, pp. 53, 2022.

[95] M. P. Uddin, M. A. Mamun, and M. A. Hossain, "PCA-based feature reduction for hyperspectral remote sensing image classification," *IETE Technical Review*, vol. 38, no. 4, pp. 377–396, 2021.

[96] E-J. Lee, Y-T. Shen, L. Pan, Z. Li, and S. S. Bhattacharyya, "DCT-based hyperspectral image classification on resource-constrained platforms," in *2021 11th Workshop on Hyperspectral Imaging and Signal Processing: Evolution in Remote Sensing (WHISPERS)*. IEEE, 2021, pp. 1–5.