Tampere University

MIR M.A.R KHAN

On Design and Optimization of Convolutional Neural Network for Embedded Systems

Tampere University Dissertations 799

Tampere University Dissertations 799

MIR M.A.R KHAN

On Design and Optimization of Convolutional Neural Network for Embedded Systems

ACADEMIC DISSERTATION
To be presented, with the permission of
the Faculty of Information Technology and Communication Sciences
of Tampere University,
for public discussion in the auditorium S2
of the Sähkötalo, Korkeakoulunkatu 3, Tampere,
on 5 May 2023, at 12 o'clock.

ACADEMIC DISSERTATION
Tampere University, Faculty of Information Technology and Communication Sciences
Finland

Responsible supervisor and Custos
Professor
Timo Hämäläinen
Tampere University
Finland

Supervisor
Docent
Heikki Huttunen
Tampere University
Finland

Pre-examiners
Professor
Guillermo Payá Vayá
Technische Universität
Braunschweig
Germany

Assistant Professor
Tomasz Kryjak
AGH University of Science
and Technology
Poland

Opponent
Senior University Lecturer
Jorma Laaksonen
Aalto University
Finland

Cover design: Roihu Inc.

ISBN 978-952-03-2891-7 (print)
ISBN 978-952-03-2892-4 (pdf)
ISSN 2489-9860 (print)
ISSN 2490-0028 (pdf)
http://urn.fi/URN:ISBN:978-952-03-2892-4

Carbon dioxide emissions from printing Tampere University dissertations have been compensated.

PunaMusta Oy – Yliopistopaino
Joensuu 2023

PREFACE

I'd like to thank Timo Hämäläinen and Jani Boutelier for supervising and guiding me throughout the research and writing the thesis. I would also like to thank Heikki Huttunen for his support and guidance.

ABSTRACT

This work presents the research on optimizing neural networks and deploying them for real-time practical applications. We analyze different optimization methods, namely binarization, separable convolution and pruning. We implement each method for the application of vehicle classification and we empirically evaluate and analyze the results. The objective is to make large neural networks suitable for real-time applications by reducing the computation requirements through these optimization approaches. The data set is of vehicles from 4 classes of vehicle types, and a convolutional model was used to solve the problem initially. Our results show that these optimization methods offer many performance benefits in this application in terms of reduced execution time (by up to 5 $\times$), reduced model storage requirements, without largely impacting accuracy, making them a suitable tool for use in streamlining heavy neural networks to be used on resource-constrained envrionments. The platforms used in the research are a desktop platform, and two embedded platforms.

CONTENTS

List of Figures

List of Tables

ABBREVIATIONS

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
cuDNN	CUDA Deep Neural Network library
GEMM	General Matrix Multiply
LBP	Local Binary Patterns

ORIGINAL PUBLICATIONS

Publication I M. Khan, H. Huttunen and J. Boutellier. Binarized Convolutional Neural Networks for Efficient Inference on GPUs. *EUSIPCO2018* (2018).

Publication II M. Khan, H. Lunnikivi, H. Huttunen and J. Boutellier. Comparing Optimization Methods of Neural Networks for Real-time Inference. *EUSPICO2019* (2019).

Publication III S. Payvar, M. Khan, R. Stahland, D. Mueller-Gritschnederand and J. Boutellier. Neural Network-based Vehicle Image Classification for IoT Devices. *IEE International Workshop on Signal Processing Systems* (2019).

Publication IV J. Boutellier, Y. Ma, J. Wu, M. Khan and S. S. Bhattacharyya. VR-PRUNE: Decidable Variable-Rate Dataflow for Signal Processing Systems. *IEEE TRANSACTIONS ON SIGNAL PROCESSING, VOL. 70, 2022* (2022).

1 INTRODUCTION

This work presents the research on optimizing neural networks and deploying them for real-time practical applications. By 'practical', we mean a setting where the neural network is running on an embedded device and is receiving data from a live camera source, and the inference is done in real time. The process starts by analyzing different optimization methods before selecting the most well-suited ones depending on the application. Once an optimization method is selected, we empirically evaluate the method by implementing it. Typically, the accuracy and the performance of the neural network is evaluated and compared with the original neural network across different platforms. Once enough empirical data is gathered, we focus on the deployment phase of the process by making it more efficient for specific hardware and the application.

The scope of this work is to explore the general advantages of optimization methods and their limits, not to satisfy a specified goal or meet certain performance requirements. This work is also not focused on finding neural network solutions, but rather use existing networks to be optimized for deployment and inference. This work also doesn't deal with safety-critical application, where the accuracy of the model is critical. Such applications also incorporate the concept of confidence in their model, which is out of scope of this work. The work is largely focused on a specific type of neural networks: *convolutional neural networks for image classification*. The optimizations are general, but are focused on deployment on embedded systems with parallel processing units.

The publications included can be categorized as follows: Specialized implementation of one optimization algorithm [33], empirical analysis and comparison of different optimization methods [34], deployment of an optimized method on an embedded system [53], and utilization of some features of the tool in high performance computing application in [4] showing versatility .

The research is mostly centered around optimizing neural networks in practical

setting, by analyze the results in performance and loss in accuracy. In the work, we also evaluate the practical benefits of these approaches. The research questions can be summarized as follows:

RQ1: What is the accuracy and performance trade-off from weight binarization, separable convolution, and weights pruning, and how they compare.

RQ2: How well the optimizations work in real-time embedded systems?

RQ3: How to speed up the process of design and deployment of neural network models?

This thesis is organized as follows: Section 2 discusses the background of neural networks and details about the algorithms, Section 3 discusses research methodology, Section 4 provides a summary of the results and their discussion, Section 5 discusses the related work, and Section 6 wraps up and concludes the thesis.

2 BACKGROUND

Machine Learning (ML) is a term that covers a wide range of approaches and methods of data analysis such as data classification, clustering and regression. Artificial Neural Networks (ANNs) fall under ML, and has a wide range of applications, that continued to expand over the past few years. ANNs are typically designed using supervised learning approaches such as the Backpropagation algorithm, where the network is initialized in a random configuration, that is then systematically and iteratively adjusted until the desired network behavior is achieved. Other approaches exist for designing ANNs, such as the works in [60][49] where neural network models are designed using genetic algorithms. Reinforcement learning is another method that can be used [62]. Neural networks have found great use in a variety of fields. Variations of ANNs have been applied in many areas such as image classification[55][58][22][31], speech recognition[18][71] [69], medical diagnosis [3][66][48][36][47], and object recognition [78][61][16][63][42]. This section provides the background for understanding how neural networks generally work and how the computations are done. Then the concept of convolutional neural networks are explained afterwards.

2.1 Perceptron

An ANN can essentially be thought of as a systen for processing information, with fundamental components known as "perceptrons"[27] (also called "neurons"). The perceptron has two basic components. First, it has a set of synapses or connecting links, each with an associated weight value as shown in Figure 2.1. The neuron computes its activation potential field v as a linear combination between input signal

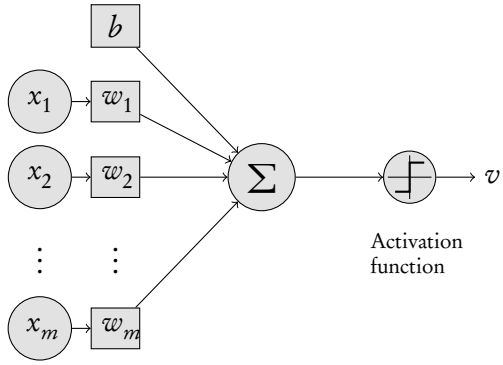


Figure 2.1 Illustration of a perceptron

x and its weights.

$$v = \sum_{i=1}^{m} w_i x_i + b \tag{2.1}$$

The parameter b in equation 2.1 is called the bias. This expression can be simplified by letting $x_0 = 1$, and $w_0 = b$, so that v can be rewritten as:

$$v = \sum_{i=0}^{m} w_i x_i \tag{2.2}$$

The second component is the activation function ϕ, which is responsible for computing the perceptron's output $y = \phi(v)$. The choice of activation function is largely dependent on the desired behavior of the neural network. A list of activation functions that are commonly used are provided below with a brief explanation.

The logistic-sigmoid is a continuous function that compresses the value of its input from 0 to 1. It can be thought of as a smooth step function.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

A plot of the sigmoid activation function is shown in Figure 2.2.

The step function is a discrete-valued function that outputs either a 0 or a 1.

$$s(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{2.4}$$

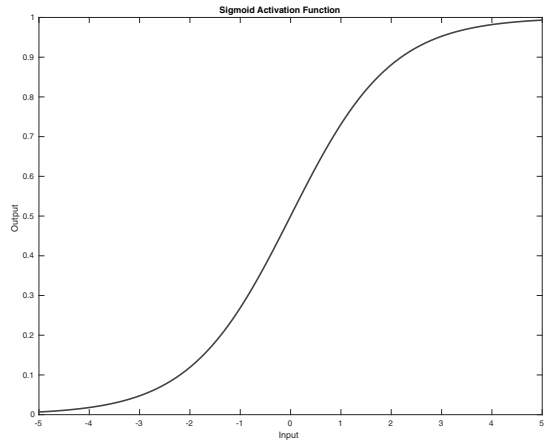


Figure 2.2 Plot of the sigmoid activation function.

A plot of the step activation function is shown in Figure 2.3.

The tanh(x) function is a bipolar version of the logistic sigmoid function

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.5}$$

A plot of the tanh activation function is shown in Figure 2.4.

Softmax can output posterior probabilities and it is usually used at the output layer classification model to provide degree of certainty of the input belonging to one of the classes.

$$\text{softmax}(x_j) = \frac{e^{xj}}{\sum m e^{x_m}} \tag{2.6}$$

ReLU is a commonly used activation function which can reduce overfitting and is given as follows.

$$\text{ReLU}(x) = \max(0, x) \tag{2.7}$$

A plot of the ReLU activation function is shown in Figure 2.5.

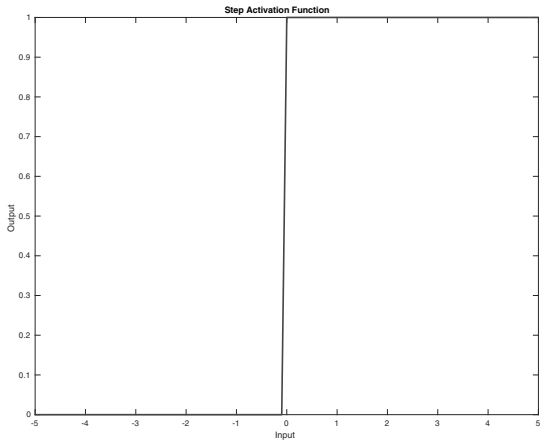


Figure 2.3 Plot of the step function.

2.2 Feed-forward neural networks

Typically a neural network is comprised of fundamental units called perceptrons. The network generally maps an input to an output that can include changes in the input's dimensionality. These days, neural networks can have multiple layers of computation before the final output layer. Between each layer, there is an activation function, a non-linearity, that can enhance the expressiveness of the network. The final layer in the network typically has a different activation function depending on the purpose. Figure 2.6 shows a neural network that receives an input $\mathbf{x} \in \mathbb{R}^4$, which is used to generate an output $\mathbf{y} \in \mathbb{R}^2$. Then essentially, the network provides a non-linear mapping from $\mathbf{x}$ to $\mathbf{y}$.

Neural networks are implemented using techniques of linear algebra due to the large number of variables involved. Computing hardware such as (Graphical Processing Units) GPUs are often optimized to handle such operations efficiently. For a single neuron, what is known as its *induced local field* can be stated as a dot product:

$$v = \mathbf{w}^T \mathbf{x} \tag{2.8}$$

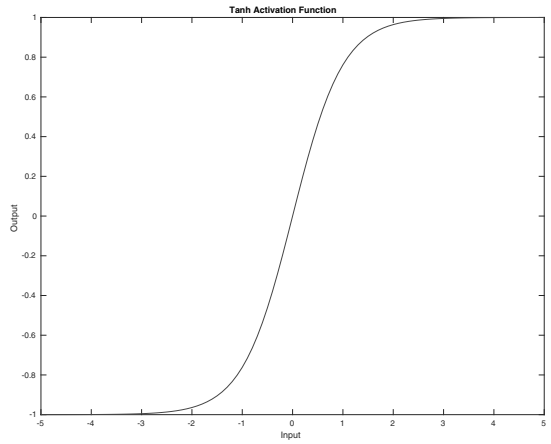


Figure 2.4 Plot of the hyperbolic tangent activation function.

Where the term $\mathbf{w} = \begin{bmatrix} w_0 & w_1 & w_2 & \dots & w_n \end{bmatrix}^T$, and $\mathbf{x} = \begin{bmatrix} x_0 & x_1 & x_2 & \dots & x_n \end{bmatrix}^T$. w_0 is the bias term for the neuron, and x_0 is set to 1. Each layer in the network can be characterized completely by its weights, or more conveniently as its weights matrix:

$$\mathbf{W} = \begin{bmatrix} w_0^0 & w_0^1 & w_0^2 & \dots & w_0^{n_h} \\ w_1^0 & w_1^1 & w_0^2 & \dots & w_1^{n_h} \\ w_2^0 & w_2^1 & w_0^2 & \dots & w_2^{n_h} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_m^0 & w_m^1 & w_m^2 & \dots & w_m^{n_h} \end{bmatrix}$$

This notation is convenient for its compact representation and usefullness when implemented in a programming language. We can interpret the number $n_h + 1$ to denote the amount of neurons in the layer h, and m represents the dimensionality of the input to the layer, which is the same as the dimensionality of the weights vector of each neuron. We can compute the output of the layer as simply the result of the

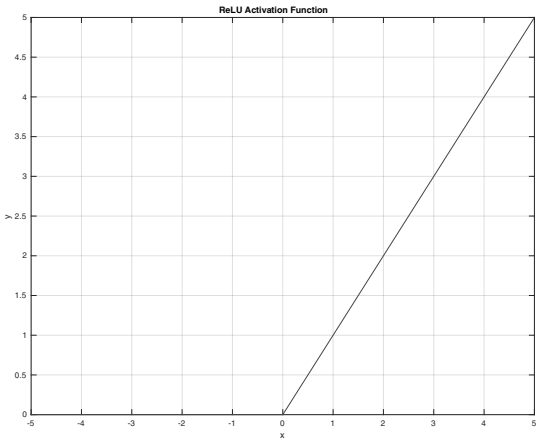


Figure 2.5 Plot of the ReLU activation function.

matrix-vector product

$$\mathbf{W}^T\mathbf{x} = \begin{bmatrix} w_0^0 & w_1^0 & w_2^0 & w_3^0 & w_4^0 \\ w_0^1 & w_1^1 & w_2^1 & w_3^1 & w_4^0 \\ w_0^2 & w_1^2 & w_2^2 & w_3^2 & w_4^2 \\ w_0^3 & w_1^3 & w_2^3 & w_3^3 & w_4^3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}.$$

When an activation function ϕ_b is applied to the matrix, it can simply be written as follows:

$$\mathbf{h} = \phi_b(\mathbf{W}^T\mathbf{x}) \tag{2.9}$$

It is common to evaluate the neural network for *batches* of inputs, especially during training. So if we have N inputs, each expressed as an m-vector in the columns of the matrix

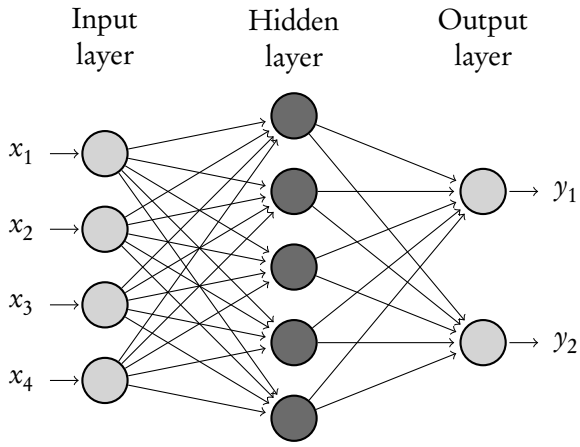


Figure 2.6 Artificial Neural Network with one hidden layer, not including bias terms.

$$\mathbf{X} = \begin{bmatrix} x_1^1 & x_1^2 & x_1^3 & \dots & x_1^N \\ x_2^1 & x_2^2 & x_2^3 & \dots & x_2^N \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_m^0 & x_m^1 & x_m^2 & \dots & x_m^N \end{bmatrix},$$

then we can represent the output matrix $\mathbf{Y}$ in a neural network with $L-1$ hidden layers, where the Lth layer represents the output layer $\mathbf{Y}$, as the following more general expression

$$\mathbf{Y} = \phi_L(\mathbf{W}_L^T \dots \phi_2(\mathbf{W}_2^T \phi_1(\mathbf{W}_1^T \mathbf{X}))\dots) \tag{2.10}$$

In order to avoid confusion, we clarify that the superscript in the elements of the matrix $\mathbf{X}$ does not denote exponentiation, but is rather an index, together with the subscripts, such that the superscript corresponds to columns, and the subscript corresponds to rows.

2.3 Convolutions

Convolution is a commonly used operation in signal processing applications, and it is defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2.11}$$

In equation 2.11, $f(\tau)$ and $g(t-\tau)$ are the input functions, also known as the kernel and the signal, respectively. The integration is carried out over all possible values of τ, from negative infinity to positive infinity. The variable t is the time parameter that specifies the point in time at which the convolution is evaluated. However, in image processing and digital systems, discrete convolution is more relevant. It is given as

$$(f * g)(n) = \sum_{-\infty}^{\infty} f[m]g[n-m] \tag{2.12}$$

In the discrete convolution equation 2.12 ,f and g are discrete functions, n is the index of the output signal, and m is the index of the input signal. More specifically, in image processing applications, the expression can be written in the 2D form:

$$y[i,j] = \sum_{m=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} h[m,n] \cdot x[i-m, j-n] \tag{2.13}$$

where, x represents the input image, h is the kernel to be convolved with the input. The indices i and j depend on the image, while m and n relate to the kernel. For example, if the size of the kernel is 5×5, then the indices m and n would range from -2 to 2. The kernel is centered around each pixel in the image at each step in the calculation. Pixel values outside the border of the kernel are considered to be 0 in most applications.

Various image processing applications can be performed by applying a convolution kernel to an image. For example, a kernel

$$\mathbf{h} = \begin{bmatrix} \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \end{bmatrix},$$

can be used to blur an image. By convolving the kernel $\mathbf{h}$ with the picture of an apple shown in Figure 2.7, results in a blurred image as shown to its right.

In convolutional neural networks, convolution filters are used as part of the learnable parameters in the network.

Figure 2.7 Image Blurring

2.4 Deep Convolutional Neural Networks

Deep Convolutional Neural Networks (CNNs) are a type of neural network that are particularly effective in image and signal processing tasks. CNNs consist of several layers, each with different types of neurons. One important layer type in CNNs is the convolutional layer, which performs a convolution operation on the input data. The goal of the convolutional layer is to identify local patterns or features in the

input data that can be used to recognize more complex patterns at higher layers.

A convolutional layer has a set of filters, also known as kernels or weights, which are learned during the training process. Each filter is a small matrix of numbers that slides over the input data, performing a dot product at each location to produce an output feature map. The filter slides over the input data with a certain stride, which is the number of pixels or units it moves at each step. The output of the convolutional layer is a stack of feature maps, where each map represents a different pattern or feature that the layer has learned to detect.

Another important layer type in CNNs is the pooling layer, which downsamples the feature maps by taking a max or average value over small regions. This helps reduce the dimensionality of the feature maps and make the network more efficient. There are also other types of layers such as activation, batch normalization, and dropout layers that are commonly used in CNN architectures.

Training a CNN involves adjusting the weights of the network to minimize a loss function, which measures the difference between the predicted output and the actual output. This is typically done using an algorithm called backpropagation, which computes the gradient of the loss with respect to the network weights and updates them in the opposite direction to minimize the loss. However, training deep CNNs can be challenging due to issues such as vanishing gradients and overfitting, which can be mitigated using techniques such as weight initialization, batch normalization, and regularization.

CNNs have been successfully applied in many areas such as image classification, object detection, and natural language processing. Some well-known CNN architectures include LeNet-5, AlexNet, VGG, GoogLeNet, and ResNet, each with their own strengths and weaknesses. Recent advances in CNNs include attention mechanisms, transfer learning, and adversarial training, which have pushed the state-of-the-art in various domains. Overall, CNNs have proven to be a powerful tool for extracting meaningful information from complex data and are likely to continue to be an active area of research in the years to come.

3 METHODOLOGY

This section discusses the research methodology, including the tools used and implementations. The mfethods we discuss include binarization, separable convolution, pruning, and a method for automating the process abstraction to modelling. The languages used in our implementations are C, OpenCL, and CUDA. The platforms used include an Intel i7-7700K + NVIDIA GTX 1080, ARM Cortex-A72 + ARM Mali T860, and a Tegra X2 (NVIDIA Jetson). The main application use case is vehicle image classification using a private dataset because of industry relevance. We summarized these points in Figure 3.1. Additionally, we list the interconnection of the papers with various elements in the research in Table 3, where each column on the right represents the published paper in the year indicated in the column. The process starts with choosing an application, then a neural network solution that is best suited for the application is selected for further optimization. Depending on the deployment platform, the appropriate optimization approaches are chosen for testing and implementation while keeping into account factors such as memory requirements, performance and accuracy loss. Each step of the diagram is discussed in further detail in later sections.

a) Method	2018	2019	2019	2022
Binarization	×	×	×	
Separable Convolution		×		
Weight Pruning		×	×	
Packing	×	×	×	
Popcount instruction	×	×	×	
Dataflow modelling + runtime				×
b) Languages				

C			×	×
OpenCL/CUDA	×	×		
c) Platforms				
NVIDIA GTX	×	×		
NVIDIA Jetson	×			
XU3				×
Mali	×	×		
RISC-V (ETISS simulator)			×	
ARM A53			×	
i7				×
d) Applications				
CNN based vehicle classification	×	×	×	×
e) Datasets				
Private data set	×	×	×	×
f) Evaluation				
Accuracy	×	×	×	
Performance	×	×	×	×
Memory	×	×	×	×
Power				×
Model complexity				×
g) Author's Contributions				
Writing the paper	×	×	×	×
Model design and training	×	×	×	×
Implementation coding	×	×	×	×
Running the test, analyzing the results	×	×		
Setting up development environment	×	×		

Table 3.1 A general outline of the elements of this work. The column date is associated with the paper published in that year.

Platform	CPU	GPU	Memory	Power
GTX	Intel i7-7700K	NVIDIA GTX 1080	32 GB DDR4	300 W
ARM	ARM Cortex-A72	ARM Mali T860	2 GB LPDDR4	15 W
TEGRA	NVIDIA Tegra X2	NVIDIA Pascal GPU	8 GB LPDDR4	15 W

Table 3.2 Technical specifications of the three platforms.

Table 3.2 shows the three platforms used in this study, along with their abbreviated names, CPU, and GPU specifications. The Desktop platform (abbreviated as GTX) consists of an Intel i7-7700K CPU and an NVIDIA GTX 1080 GPU. The CPU has four physical cores and eight threads, with a base clock speed of 4.2 GHz and a maximum turbo frequency of 4.5 GHz. The GPU has 2560 CUDA cores and 8 GB of GDDR5 memory, with a base clock speed of 1607 MHz and a boost clock speed of 1733 MHz.

Thee embedded ARM system (abbreviated as ARM) is a low-power platform designed for use in mobile and embedded devices. It features an ARM Cortex-A72 CPU and an ARM Mali T860 GPU. The CPU has four cores, with a maximum clock speed of 2.5 GHz. The GPU has 16 shader cores and supports OpenGL ES 3.2, OpenCL 1.2, and Vulkan graphics APIs.

The embedded NVIDIA Jetson system (abbreviated as TEGRA) is another low-power platform designed for use in embedded devices. It features an NVIDIA Tegra X2 CPU and an NVIDIA Pascal GPU. The CPU has eight cores, with a maximum clock speed of 2.0 GHz. The GPU has 256 CUDA cores and 4 GB of LPDDR4 memory, with a base clock speed of 1302 MHz and a boost clock speed of 1455 MHz.

3.1 Application

The application chosen in this work is vehicle classification, and this area of application was chosen for various reasons such as the practical nature of the problem and as a continuation of on going research in the university in collaboration with industry. This particular application is highly industry relevant and is heavily utilized in com-

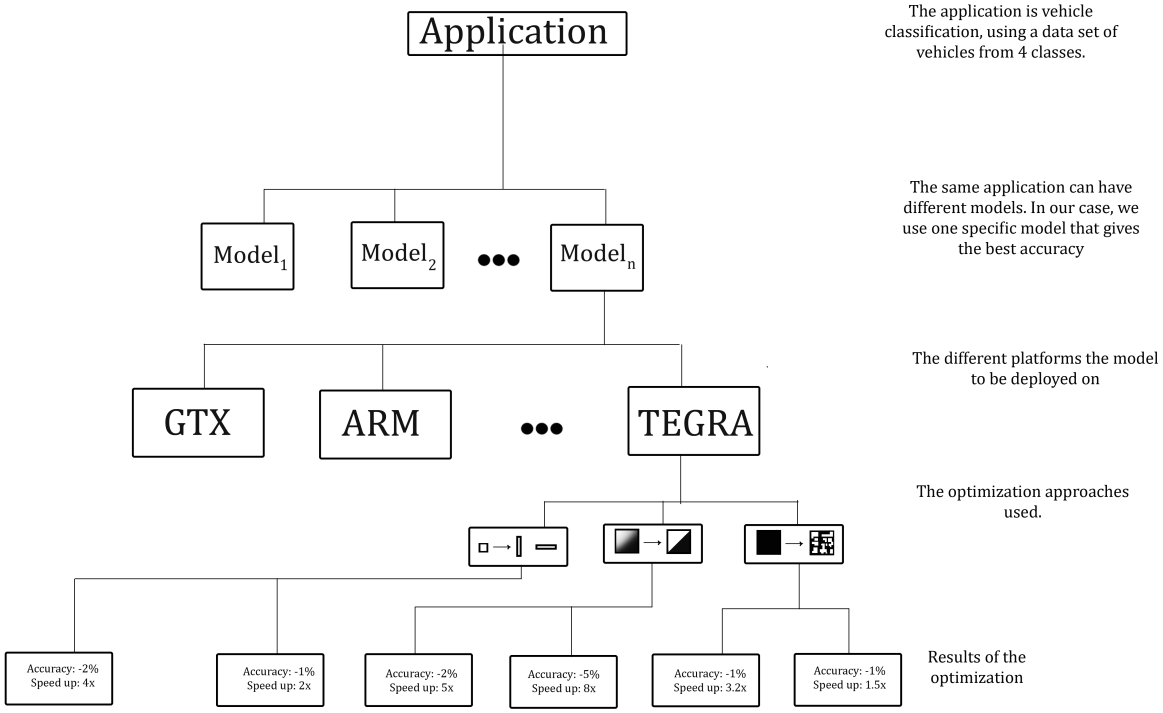


Figure 3.1 Visual outline of the research methodology

panies. A neural network solution is already proposed for the problem in [28], however, performance limitations led to further research for optimizing the solution for embedded devices [75]. We carry on and further optimize the problem. Generally, neural networks are known for being very redundant [29][9][21][64]. Many techniques exist in order to optimize various aspects of neural networks, and they can be categorized into several groups: pruning of parameters [23][23][15], quantization of weights [82][74][7], reduced precision weights[81][13][19][70], low-rank approximations [72][12][39][38], compact convolution kernels [46][80][57][25], and distillation [79][77][8][76][45].

The goal of such optimization approaches is to search for alternative solutions that preserve accuracy in a neural network while reducing utilization of resources such as memory and computation. In this work, we apply a selection of optimization methods, namely binarization, separable convolution, and pruning to this problem in present and analyze the results. Performance measurements are made on different

platforms, summarized below. Details are provided in later sections:

- Intel i7-7700K + NVIDIA GTX 1080
- ARM Cortex-A72 + ARM Mali T860
- Tegra X2 (NVIDIA Jetson)

Throughout the research, several standard datasets such as the MNIST and Cifar10 were used to verify the optimization approaches. Often, the papers that propose these methods have carried out their experiments on these standard datasets, allowing for direct comparison and verification of our implementation before applying it in other unexplored areas.

3.2 Author's contributions

The contributions of this work includes implementing binarized convolutional neural networks and evaluating the performance on GPUs, empirical comparison of several optimization methods, including binarization, separable convolution, and pruning. Additionally, a dataflow model for automating the implementation of neural networks is presented.

Below is a detailed list of the roles of each of the authors in the publications, ordered by publication date, from oldest to recent.

1. Mir Khan came up with the idea for the research, planned and implemented all versions of the neural networks, ran the experiments and analyzed the results. Heikki Huttunen participated in the planing of the neural network and along with Jani Boutellier, reviewed the implementations and the results. While Mir Khan wrote the majority of the paper, Heikki Huttunen and Jani Boutellier also participated in writing, editing, and reviewing of the paper.
2. Mir Khan, Henri Lunnikivi, and Jani Boutellier came up with the idea for the research. Both Mir Khan and Henri Lunnikivi implemented the different versions of all the optimization approaches, ran the experiments, analyzed the results, and wrote most of the paper. Mir Khan's focus was on binarization and separable convolution, while Henri Lunnikivi focused on pruning and separable convolution. Heikki Huttunen and Jani Boutellier also participated in reviewing the results and editing and reviewing the manuscript.
3. Saman Payvar, Jani Boutellier, Rafael Stahl, and Daniel-Gritschender came up

with the idea for the research, and along with Mir Khan, were all involved in the development process. Saman Payvar ran most of the experiments and analyzed the results, while Mir Khan planned and implemented the binarized image classifier for the experiments and analyzed parts of the results. Majority of the paper was written by Saman Payvar , while Mir Khan, Jani Boutellier, Rafael Stahl, and Daniel-Gritschender were also heavily involved in writing, editing and reviewing the manuscript and reviewing of the results of the experiments.

4. Jani Boutellier came up with the idea for the research, and he, along with Yujunrong Ma, Jiahao Wu, and Mir Khan were involved in the development process. Mir Khan's focus was on integrating and interfacing with the neural networks, and planning and implementing the neural networks. Majority of the paper was written by Jani Boutellier, while Yujunrong Ma, Jiahao Wu, Mir Khan, and Shuvra S. Bhattacharyya were also involved in the writing, editing and reviewing of the manuscript.

3.3 Neural network

We should emphasize that the focus of this research is not centered on finding neural network solutions for problems, but rather we take existing solution from other works, and then solve problems related to performance and deployment in resource-constrained environments such as mobile and embedded devices. The contributions of this works are mostly about extending neural network optimization approaches to specific application areas and neural network solutions. These contributions are highlighted in the next sections where our implementations are discussed in detail. Optimization approaches offer different benefits, for example, binarization can dramatically reduce the model's storage requirements depending on the choice of packing bitwidth, e.g., by up to 32X for a packing bitwidth of 32. On CPU platforms, where the model computes sequentially and assuming no parallelism, a similar speed up can be achieved. On GPU, the speed up is slightly lower due to parallelism. Binarization however results in the most reduction in accuracy. Pruning has the most significant impact on model size compression. Separable convolution offers a balance of benefits.

The results of paper [28] were reproduced and verified before any optimization

approaches were applied. The original network has 5 layers, starting with 2 convolutional layers, then followed by 2 dense layers. The convolutional layers use 5×5 filters, and each result in 32 featuremaps. The first dense layer receives the output of the final convolutional layer, and results in 100 outputs. The last layer before the final classification layer also results in 100 outputs. We perform the training in Tensorflow [2]. The weights are initialized according to [17]. We use ReLU [1] activations throughout the network and an RMSprop optmizer [24]. The experimental optimization approaches were first applied in Tensorflow and the accuracy results evaluated. Then the research goes to the next phase where the new version of the network is implemented in a number of different platforms using OpenCL and CUDA, and their execution time is evaluated on each platform.

3.4 Optimization Challenges

Convolution is a key operation and an important algorithm in image processing applications; the problem is that it is resource intensive and slow on CPUs. Although a large number of dedicated hardware exists for deep learning as presented in [5][67][6], many limitations are still present. The most commonly used approach for speeding up convolutions is to reduce the process into a matrix multiplication problem, which GPUs are optimized to handle. However, it is still very expensive to form the matrix that is then used for multiplication for computing the convolution.

One approach to reduce the computation cost is using separable convolution or low-rank approximations, where each convolution operation is replaced by a combination of two smaller convolutions, resulting in a reduced total number of operations and memory requirements. More precisely, a $k \times k$ convolution is replaced by a $1 \times k$ convolution, followed by another $k \times 1$ convolution. This approach is essentially expected to reconstruct the results of the full convolution through this cheaper process. Each smaller convolution has intermediate featuremaps that can be tuned for maximizing the balancing between performance and accuracy. This process is briefly illustrated in Figure 3.2.

Another variation of separable convolution exists called depthwise separable convolution and it's used heavily in MobileNet [25]. Instead of low rank separability, the separability is in the operations performed, where a regular convolution is re-

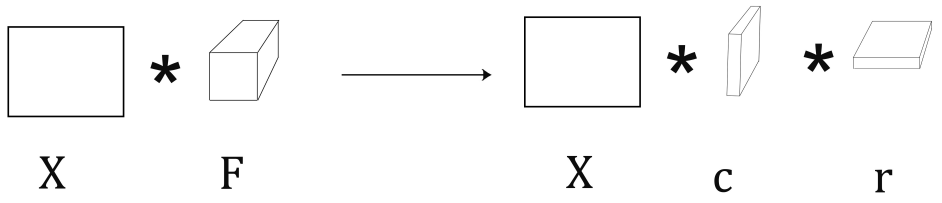


Figure 3.2 The principle of separable convolution, compared with regular convolution.

placed by a convolution applied separately per channel, followed by a 1x1 convolution, resulting in less computation. More precisely, a depthwise convolution of a $h \times w \times c$ tensor with a $k \times k$ kernel performs the convolution for each pixel with the $k \times k$ kernel, but a separate set of $1 \times 1 \times c$ kernels tensor is used to combine the channels and compute the featuremaps. This process is illustrated briefly in Figure 3.3, where an image with 3 channels is convolved with a set of filters with the same number of channels. However, the results (in blue, with the number of channels unchanged) are not combined as it is typically done in convolutions in neural networks. Instead, a separate set of filters with kernel size 1×1 is convolved with the result and combined as usual. Then the result is shown in Green on the right.

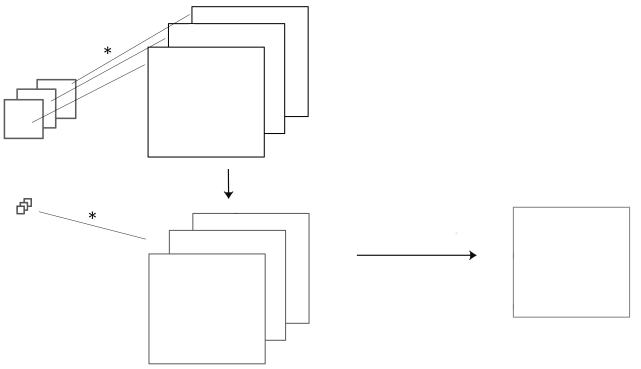


Figure 3.3 Depthwise separable convolution, illustrated for one output featuremap.

Binarized neural networks is an approach where the weights and activations for intermediate computations are binarized to +1 and -1. For the MNIST data set a 7x speed up can be achieved [26]. Substantial speed up can be shown in CPU im-

plementation and evaluated on the ImageNet dataset by packing 1-bit weights into 32-bit words, enabling replacement of multiplication operations by logic XNORs. For a simple dot product this is shown as follows

$$\mathbf{a}\cdot\mathbf{b} = \mathtt{W} - 2 \times \mathtt{popcount}(\mathtt{xor}(A,B)), \tag{3.1}$$

In this research, we apply the binarized neural network approach and implement it for GPUs. At the time of this research, in this work, we obtained the first performance advantages offered by (Binarized convolutional neural networks) BCNN on GPUs. Additionally, we explored different approaches for the binarization of the first layer in the network, which is typically not binarized due to the increased reduction in accuracy. The approaches we explored include transforming the input using Local Binary Patterns (LBP) [51], thresholding on a grayscale input, and thresholding on a color input. Threshholding basically means that we choose a pixel value in the image, and all pixels above this value are set to 1, and the ones below it are set to 0. The results are presented and discussed in Section 4.

Pruning is a method for reducing the number of weights in a network in the fully-connected layers resulting in a sparse weights matrix, which allows the computation to be performed more efficiently using sparse matrix-vector multiplication. Pruning is a general approach where a criteria is used to eliminate weights with certain characteristics from the network by setting them to 0. In this work, the relevant approach to fix weights to 0 if they fall below the threshold $T = \frac{max(W^l)+min(W^l)}{2}$. This can result in a very large drop in the model's accuracy at this stage; however, by retraining the network for several epochs, the accuracy can largely be restored. Some implementations of this method incorporates L2 regularization in the initial training stage, which further increases the sparsity in the layers. The process is illustrated as pseudocode in Algorithm 1.

One of the advantages of this approach is the massive reduction in memory requirements. While there are no significant performance advantages on GPUs, there are techniques such as sparse matrix-vector multiplication that leverages the sparsity of the matrix to reduce the number of computations.

Algorithm 1 Fully-connected layer pruning procedure

for l in Layers **do**
    for w in W^l **do**
        $w \leftarrow$ LoadPretrained(w)
for stage in Stages **do**
    // Retrain with L2 regularization
    for epoch in Train_Epochs **do**
        for l in Layers **do**
            for w in W^l **do**
                $w \leftarrow \text{Update}(w, \eta, \frac{\partial \mathscr{L}(\mathbf{X})}{\partial w})$
    // Pruning stage
    for l in DenseLayers **do**
        $T \leftarrow \frac{max(W^l)+min(W^l)}{2}$
        for w in W^l **do**
            if $w < T$ **then**
                $w \leftarrow 0$
    // Fine-tune parameters
    for epoch in Tuning_Epochs **do**
        for l in Layers **do**
            for w in NonZero(W^l) **do**
                $w \leftarrow \text{Update}(w, \eta, \frac{\partial \mathscr{L}(\mathbf{X})}{\partial w})$

Figure 3.4 From left to right, a 'bus', 'normal car', 'truck', and a 'van'.

3.5 Dataset

The most commonly used dataset in this work is the private cars dataset obtained by a courtesy of a company working in this field. The data set has 6555 images. Each image is 96x96 32-bit color images around size of 108 KB. There are 4 categories of cars: bus, normal car, van, and truck, as illustrated in Fgure 3.4. The images have been taken by a camera in various weather and lighting conditions. The data is split into 80% for training, 10% for validation, and 10% for testing.

3.6 Implementation

In this section, we discuss the implementation stage of the research. After the neural network solution, target platforms, and the desired optimization approaches have been identified, the network is implemented in Tensorflow for training from scratch according to the specifications in the paper related to the network in question. Then this network is preserved for later verification before an optimized version of the network is implemented in Tensorflow and the accuracy is verified to be at least as good as with the first implementation. Once the correctness is verified, we proceed to implement the OpenCL/CUDA version.

After this step, a CUDA version is implemented. The implementation is verified again for correctness and then performance measurements are taken. This is done first on the desktop machine, and then later compiled and tested on the target platforms. If the target platform does not have a CUDA capable device, we re-implement the program in OpenCL. This process is very simple and can even be automated, as it mostly involves replacing terms in the kernel code from CUDA terms to OpenCL. While the host code is not as simple to translate, it is generic and involves little changes.

We also re-implement the original version of the network and the optimized version using vendor libraries, which claim to be optimized for the target platform. This is done in order to compare the performance of the vendor library implementation with our implementation, to insure that our implementation does not suffer from basic optimization issues. Additionally, we wish to know to full extent of performance benefits achieved from the neural network optimization approach applied.

These steps in the process be can summarized as follows:

1. The neural network is implemented in Tensorflow according to the specifications in the paper [28] and verified.
2. The network is reimplemented in Tensorflow with the optimization. Any parameters that are a part of the optimization approach are selected in this stage.
3. The new optimized version of the neural network is then implemented in OpenCL and CUDA in order to run on the target platforms. For performance comparison, we also implement this version of the network using vendor li-

braries such as ArmCL, cuDNN, and cuBLAS.

4 SUMMARY OF RESULTS

In this section, the results are summarized by optimization techniques.

4.1 Binarization

Table 4.1 shows the impact of input binarization on accuracy. It is typical in binarized neural networks to leave the first layer (the input layer) in full-precision, and binarize all subsequent layers. The last row in the table shows the accurcy of the network without any binarization. The rows 1 to 3 showcase different techniques of input binarization and how each method affects the accuracy. The 4th row is the result with only the first layer binarized.

Table 4.2 shows the speed-up achieved from binarization per layer on the GTX1080 platform. The speed-up expressed in the last row is relative to equivalent implementations with cuDNN. It should be noted that the equivalence is in terms of network accuracy performance, and not in terms of computation results, so in this case, even though the results from cuDNN and binarization may not match exactly, the final outputs of the neural network are largely similar as it pertains to classification accu-

Table 4.1 Impact of different input-binarization schemes on classification accuracy

Method	Accuracy
LBP	92.06%
Thresholding Grayscale	89.16%
Thresholding RGB	92.52%
No input binarization	94.20%
Full-precision network	97.09%

racy.

We can notice from Table 4.2 that the largest speed up is achieved in the final layer, where the layer is basically a matrix-vector multiplication. A speed-up of roughly 32× is in-line with the results presented in the original binarized neural networks paper [26]. However, in the matrix-multiplication stage of convolution layers, the 32 × speed up was not achievable, mostly due to the overhead of packing the output of the preceding im2col3d matrix.

It is common to decompose convolutions on GPUs into two steps where the input is first rearranged (im2col) and then the convolution result is computed through a simple matrix-matrix multiplication. Im2Col3d in the first row refers to the 3d version of the im2col algorithm, rearranging an $M \times N \times C$ matrix into $P \times (M \times N)$ matrix, where $P \in \mathbb{R}^{C \times K \times K}$, for a kernal of size $K \times K$.

GEMM-convolution refers to General Matrix-matrix multiplication, which computes the final results of the convolution. The layer that follows is Max-Pooling, where a running window of size $k \times k (k = 2)$ in our case, moves throughout the image, and the maximum value in each window is retained while the other values are discarded. This reduces the dimensions of the image by k in each dimension. The final row represents the final layer in the network before the classification results are computed. This is simply a matrix-vector multiplication procedure.

Table 4.3 shows the overall network speedup on 3 platforms, GTX1080, Mali T860 and the Tegra X2. We compare the execution time on each platform, implemented using the optimized vendor library and our implementation. cuDNN is the NVIDIA library for implementing neural networks. Arm CL is the vendor provided library for ARM platforms, and BCNN is our implementation. Additionally, we compare the performance with an additional step of optimization (binarized inputs) in the last row.

4.2 Separable Convolution

Table 4.4 compares the various advantages achieved through binarization, spearable convolution, and pruning, and how these methods compare with each other. The first two columns shows the implementations done with vendor libraries, which are expected to be optimal. The last two columns is our implementation. The GTX label refers to a desktop machine equipped with a GTX 1080 GPU, Mali refers to an

Table 4.2 Runtime per-layer (GTX1080), in the order of their execution

Layer	cuDNN	Binarized	Speed-up
Im2col3d $(96, 96, 3)$	21.63 μs	3.17μs	6.82×
GEMM-convolution $(32, 5, 5, 3)$	37.54μs	8.61μs	4.36×
Max-Pooling $(96, 96, 32)$	5.22μs	8.26μs	0.63×
Im2col3d $(48, 48, 32)$	65.41μs	5.50μs	11.89×
GEMM-convolution $(32, 5, 5, 32)$	69.28μs	8.10μs	8.55×
Max-Pooling (48,48,32)	5.38μs	2.66μs	2.02×
Fully-Connected $(100, 24 \times 24 \times 32)$	200.03μs	6.28μs	31.85×

Table 4.3 Runtime of the network on each platform

Implementation Method	GTX1080	Mali T860	Tegra X2
cuDNN (full-precision)	401.83μs	N/A†	2.27 ms
Arm CL (full-precision)	N/A†	29.61 ms	N/A†
BCNN	102.39μs	23.63 ms	0.53 ms
BCNN with binarized inputs	**55.63**μs	**17.58** ms	**0.41** ms

†Library not compatible with this platform.

embedded platform FireFly. The last 4 rows show the accuracy, memory requirements, number of multiplications, and number of additions for the whole network. It can be noted from the table that the biggest reduction in memory requirements is achieved with pruning, while the largest reduction in execution time is obtained through binarization.

Figure 4.1 illustrates how the number of intermediate feature maps K in separable convolution affects the accuracy. The red dot corresponds to 7 intermediate featuremaps, which is our chosen value for the vehicle classifer implementation. This

Table 4.4 Results of optimization methods at inference

	Baseline	Pruning	SepConv	BCNN
GTX	3.1 ms	0.7 ms	1.1 ms	**0.06** ms
Mali	120 ms	66 ms	80 ms	**17.6** ms
Accuracy	97.5%	95.5%	97.3%	94%
Memory (KB)	7350	**150**	7254	230
Muls (in 10^6)	82.95	81.53	18.30	**0.43**
Adds (in 10^6)	86.12	84.35	20.37	**0.43**

value was chosen because it offers the best trade off between performance and accuracy. Even though a slightly higher accuracy can be achieved later at around K=12 as it appears in the graph, it comes at an increased cost in performance is clearly visible from the graph.

We also illustrate in Figure 4.2 how the loss in the neural network is affected by changing K. We can clearly see that higher values of K yields better accuracy performance, but this advantage diminishes at values close to K=9 at the bottom of the graph.

4.3 Pruning

Figure 4.3 illustrates the process of pruning and how it impacts accuracy throughout the training and retraining phases. The level of sparsity throughout this process is also shown as dashed lines, with the range corresponding to the right vertical axis. The accuracy of the network starts to vacillate within a bigger range; however, an accuracy of 97% can be achieved at nearly 99% sparsity.

The green plot shows the *fine-tuning* stage, where the pruned weights are not up-

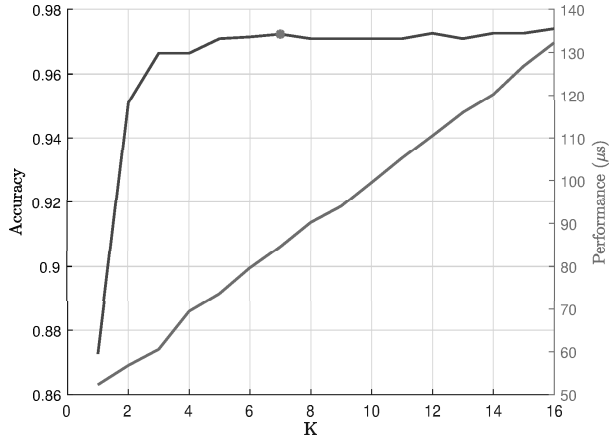


Figure 4.1 Performance/accuracy trade-off for different values of K.

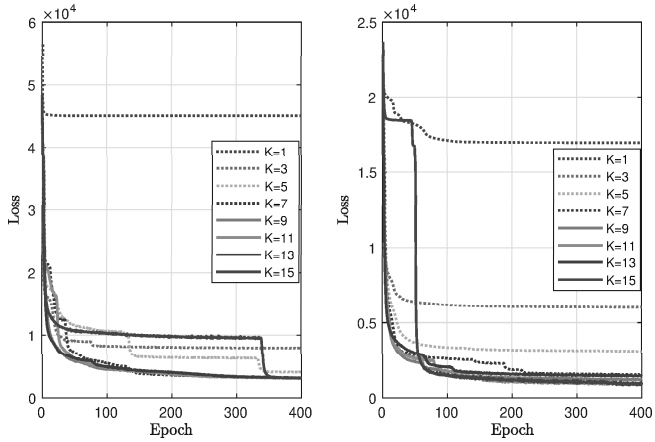


Figure 4.2 Loss for convolutional layers 1 (left) and 2 (right) shown for different values of K.

dated and held at 0. The blue plot marks the stage where all the weights are updated and the previously pruned weights in the previous step are restored and re-updated. We report the accuracy on the validation set in Figure 4.3.

4.4 Conclusion of Results

Our results show that these optimization methods offer many benefits. The optimization method needs to be chosen depending on the optimization objective. For example, separable convolution can offer minor reduction in model size, and a mod-

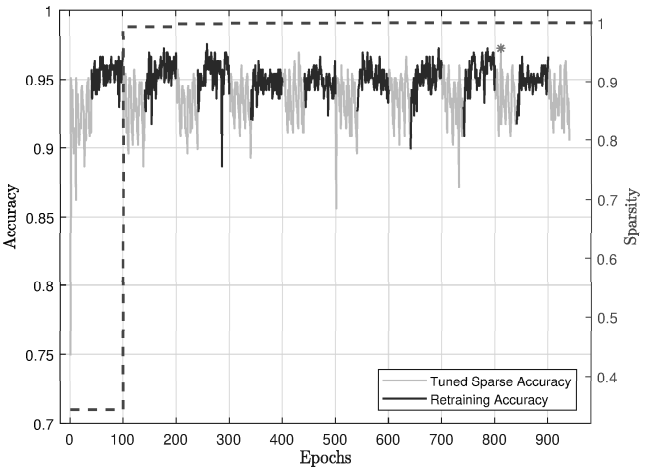


Figure 4.3 Accuracy of the network on the validation set at retraining stages and fine-tuning stages.

est improvement in performance, and pruning can dramatically reduce the model size. While this reduces the number of operations and can offer benefits on CPUs, it is difficult to leverage this on GPUs without using specialized techniques of sparse matrix-multiplication, which have an overhead of processing the matrix. Binarization can both reduce the model size and improve the performance. The loss in accuracy is another thing to keep in mind when choosing an optimization method. For example, binarization can result in a large reduction in accuracy in some applications, but in other applications like the MNIST dataset where the images are black and white and the features are clear, the reduction in accuracy is very small, making binarization very suitable for this particular application.

We restate the three research questions:

RQ1: What is the accuracy and performance trade-off from weight binarization, separable convolution, and weights pruning, and how they compare.

RQ2: How well the optimizations work in real-time embedded systems?

RQ3: How to speed up the process of design and deployment of neural network models?

The dissertation has successfully answered the three research questions posed in the study. Firstly, regarding RQ1, the research examined the accuracy and performance trade-off resulting from weight binarization, separable convolution, and weight pruning techniques. The study employed a comprehensive evaluation framework to compare the three optimization techniques and determine their effective-

ness. The results indicated that weight binarization and separable convolution were more effective in reducing the model size while maintaining accuracy, whereas weight pruning offered better memory cost reduction. Secondly, the dissertation addressed RQ2 by evaluating the effectiveness of the optimization techniques in real-time embedded systems. The research employed a benchmarking methodology that assessed the performance of the models in various real-time scenarios. The results showed that the optimized models outperformed their non-optimized counterparts and could even achieve real-time performance on resource-constrained embedded devices. Lastly, RQ3 was answered by proposing a novel methodology that speeds up the process of designing and deploying neural network models. The methodology involved the use of automated tools that streamlined the optimization process, reducing the time and effort required for model design and deployment. The study demonstrated the effectiveness of the proposed methodology by evaluating its performance in several benchmark datasets and comparing it to existing approaches.

5 RELATED WORK

This work is mainly concerned with vehicle classification using neural networks and optimizing them for real-time inference. Many works show that neural networks are excellent at image classification applications [68][56][50][43][11], object recognition [37][59][68][73][41], and object detection [52][14][44].

Binarized neural networks is a method of optimizing neural networks for inference by reducing their size and number of computations. Some of the earliest works in BNNs were introduced in [35] [26] and their performance was demonstrated on the MNIST dataset on GPUs, and later it was extended [10] to the training phase of the neural network. The work in [54] demonstrated the performance advantages of the method on CPUs, which can result in up to 32 $\times$ model size reduction and speed up. This work extended binarized neural networks to convolutional neural networks and was the first to demonstrate the performance of binarized convolutional neural networkss on GPUs in the use case of vehicle classification.

Separable convolution is a technique for reducing the model size and computation in a neural network by replacing convolutions with two separable convolutions [30]. A similar technique later appeared [25] [57] with a slightly modified approach that results in less computation, and is heavily used in MobileNet and its variants. This work demonstrated it works for detailed image classification applications such as vehicle classification.

Another approach for optimizing neural network inference is pruning, where a proportion of the parameters in the network are removed. The method reduces the parameters at very little loss of accuracy, and can have some minor performance advantages using sparse matrix-vector multiplication algorithms. These approaches largely have benefits in model size reduction. While any variations of this approach exist, the main difference is in the pruning criteria. Earlier pruning techniques simply removed weights that are low in magnitude, but more sophisticated approaches evolved later. A method was discussed in [40] where the second derivative of the loss

function used as the pruning criteria, their results show a reduction of model size by up to 4 times, with a slight increase in accuracy speed. More techniques emerged where the pruning criteria is closely connected with the loss function [32], and some approaches focused on pruning by removing individual neurons rather than the individual connections [65], or combining several different methods [20] for achieving better pruning results. Our results show that in the use case of vehicle classification, this method can offer great benefit in reducing the model size with low loss in accuracy; however, this does not translate to significantly reduced computation time on GPUs.

6 CONCLUSION

We presented our optimized implementations of a neural network for vehicle classification. Multiple optimization approaches have been evaluated on multiple platforms, and their impact on performance, and model size have been analyzed thoroughly. Our work shows that the best performance can be achieved through binarization, albeit at a greater loss in accuracy. However, in real-time applications, it could be more beneficial to classify multiple images over a period of time with low accuracy per classification, than classifying less images with higher classification accuracy. These optimization approaches are able to bring performance of such neural networks closer to real-time, making them suitable for such applications. A dataflow model has also been introduced for simplifying the process of implementing the neural network, paving the way for applying such optimizations automatically in the future.

In future work, we aim to expand on the generality of our results by testing our optimization approaches on a larger set of datasets and neural network architectures. This will allow us to assess the scalability of our approaches and identify any limitations in their applicability. Moreover, we plan to shift our focus towards automated optimizations, which could significantly reduce the time and effort required to optimize neural networks manually. To this end, we intend to explore high-level abstraction models and tools that can automate the optimization process based on user-defined constraints and objectives. This will enable us to efficiently explore the optimization space and identify the best optimization approaches for a given neural network and platform combination.

REFERENCES

[1] A. F. Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375* (2018).

[2] E. Allaire. *TensorFlow*. 2016.

[3] F. Amato, A. López, E. M. Peña-Méndez, P. Vaňhara, A. Hampl and J. Havel. *Artificial neural networks in medical diagnosis*. 2013.

[4] J. Boutellier, Y. Ma, J. Wu, M. Khan and S. S. Bhattacharyya. VR-PRUNE: Decidable Variable-Rate Dataflow for Signal Processing Systems. *IEEE TRANSACTIONS ON SIGNAL PROCESSING, VOL. 70, 2022* (2022).

[5] Y.-H. Chen, T. Krishna, J. S. Emer and V. Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52.1 (2017), 127–138. DOI: 10.1109/JSSC.2016.2616357.

[6] Y.-H. Chen, T.-J. Yang, J. Emer and V. Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), 292–308.

[7] S. Chen, W. Wang and S. J. Pan. Metaquant: Learning to quantize by learning to penetrate non-differentiable quantization. *Advances in Neural Information Processing Systems* 32 (2019).

[8] Y. Chen, N. Wang and Z. Zhang. Darkrank: Accelerating deep metric learning via cross sample similarities transfer. *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.

[9] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary and S.-F. Chang. An exploration of parameter redundancy in deep networks with circulant projections. *Proceedings of the IEEE international conference on computer vision*. 2015, 2857–2865.

[10] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[11] C. Deng, X. Liu, C. Li and D. Tao. Active multi-kernel domain adaptation for hyperspectral image classification. *Pattern Recognition* 77 (2018), 306–315.

[12] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. *Advances in neural information processing systems* 27 (2014).

[13] S. Ge. Efficient deep learning in network compression and acceleration. *Digital Systems*. IntechOpen London, UK, 2018.

[14] S. Ge, J. Li, Q. Ye and Z. Luo. Detecting masked faces in the wild with lle-cnns. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, 2682–2690.

[15] S. Ge, Z. Luo, S. Zhao, X. Jin and X.-Y. Zhang. Compressing deep neural networks for efficient visual inference. *2017 ieee international conference on multimedia and expo (icme)*. IEEE. 2017, 667–672.

[16] M. Ghifary, W. B. Kleijn and M. Zhang. Domain adaptive neural networks for object recognition. *Pacific Rim international conference on artificial intelligence*. Springer. 2014, 898–904.

[17] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, 249–256.

[18] A. Graves and N. Jaitly. Towards end-to-end speech recognition with recurrent neural networks. *International conference on machine learning*. PMLR. 2014, 1764–1772.

[19] S. Gupta, A. Agrawal, K. Gopalakrishnan and P. Narayanan. Deep learning with limited numerical precision. *International conference on machine learning*. PMLR. 2015, 1737–1746.

[20] S. Han, H. Mao and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[21] S. Han, J. Pool, J. Tran and W. Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems* 28 (2015).

[22] T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie and M. Li. Bag of tricks for image classification with convolutional neural networks. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, 558–567.

[23] Y. He, X. Zhang and J. Sun. Channel pruning for accelerating very deep neural networks. *Proceedings of the IEEE international conference on computer vision*. 2017, 1389–1397.

[24] Hinton. Divide the gradient by a running average of its recent magnitude. *Lecture 6.5-rmsprop: . COURSERA: Neural networks for machine learning*. COURSERA. 2012.

[25] 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).

[26] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv and Y. Bengio. Binarized neural networks. *Advances in neural information processing systems* 29 (2016).

[27] E. Hunt. *Perceptrons*. 1971.

[28] H. Huttunen, F. S. Yancheshmeh and K. Chen. Car type recognition with deep neural networks. *2016 IEEE intelligent vehicles symposium (IV)*. IEEE. 2016, 1115–1120.

[29] Y. Izui and A. Pentland. Analysis of Neural Networks with Redundancy. *Neural Computation* 2.2 (1990), 226–238. DOI: 10.1162/neco.1990.2.2.226.

[30] M. Jaderberg, A. Vedaldi and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866* (2014).

[31] N. Jmour, S. Zayen and A. Abdelkrim. Convolutional neural networks for image classification. *2018 international conference on advanced systems and electric technologies (IC_ASET)*. IEEE. 2018, 397–402.

[32] E. Karnin. A simple procedure for pruning back-propagation trained neural networks. *IEEE Transactions on Neural Networks* 1.2 (1990), 239–242. DOI: 10.1109/72.80236.

[33] M. Khan, H. Huttunen and J. Boutellier. Binarized Convolutional Neural Networks for Efficient Inference on GPUs. *EUSIPCO2018* (2018).

[34] M. Khan, H. Lunnikivi, H. Huttunen and J. Boutellier. Comparing Optimization Methods of Neural Networks for Real-time Inference. *EUSPICO2019* (2019).

[35] M. Kim and P. Smaragdis. Bitwise neural networks. *arXiv preprint arXiv:1601.06071* (2016).

[36] H. Kordylewski, D. Graupe and K. Liu. A novel large-memory neural network as an aid in medical diagnosis applications. *IEEE Transactions on Information Technology in Biomedicine* 5.3 (2001), 202–209.

[37] A. Krizhevsky, I. Sutskever and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM* 60.6 (2017), 84–90.

[38] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets and V. Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553* (2014).

[39] V. Lebedev and V. Lempitsky. Fast convnets using group-wise brain damage. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, 2554–2564.

[40] Y. LeCun, J. Denker and S. Solla. Optimal brain damage. *Advances in neural information processing systems* 2 (1989).

[41] J. Li, X. Liu, M. Zhang and D. Wang. Spatio-temporal deformable 3d convnets with attention for action recognition. *Pattern Recognition* 98 (2020), 107037.

[42] M. Liang and X. Hu. Recurrent convolutional neural network for object recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, 3367–3375.

[43] L. Liu, P. Fieguth, Y. Guo, X. Wang and M. Pietikäinen. Local binary features for texture classification: Taxonomy and experimental study. *Pattern Recognition* 62 (2017), 135–160.

[44] A. T. Lopes, E. De Aguiar, A. F. De Souza and T. Oliveira-Santos. Facial expression recognition with convolutional neural networks: coping with few data and the training sample order. *Pattern recognition* 61 (2017), 610–628.

[45] Y. Luo, A. Chen, K. Yan and L. Tian. Distilling self-knowledge from contrastive links to classify graph nodes without passing messages. *arXiv preprint arXiv:2106.08541* (2021).

[46] N. Ma, X. Zhang, H.-T. Zheng and J. Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. *Proceedings of the European conference on computer vision (ECCV)*. 2018, 116–131.

[47] G. Marques, D. Agarwal and I. de la Torre Dıez. Automated medical diagnosis of COVID-19 through EfficientNet convolutional neural network. *Applied soft computing* 96 (2020), 106691.

[48] S. Moein. *Medical diagnosis using artificial neural networks*. IGI global, 2014.

[49] D. J. Montana, L. Davis et al. Training feedforward neural networks using genetic algorithms. *IJCAI*. Vol. 89. 1989, 762–767.

[50] K. Nogueira, O. A. Penatti and J. A. Dos Santos. Towards better exploiting convolutional neural networks for remote sensing scene classification. *Pattern Recognition* 61 (2017), 539–556.

[51] T. Ojala, M. Pietikäinen and D. Harwood. A comparative study of texture measures with classification based on featured distributions. *Pattern recognition* 29.1 (1996), 51–59.

[52] J. Pang, K. Chen, J. Shi, H. Feng, W. Ouyang and D. Lin. Libra r-cnn: Towards balanced learning for object detection. *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, 821–830.

[53] S. Payvar, M. Khan, R. Stahland, D. Mueller-Gritschnederand and J. Boutellier. Neural Network-based Vehicle Image Classification for IoT Devices. *IEE International Workshop on Signal Processing Systems* (2019).

[54] M. Rastegari, V. Ordonez, J. Redmon and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *European conference on computer vision*. Springer. 2016, 525–542.

[55] W. Rawat and Z. Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural computation* 29.9 (2017), 2352–2449.

[56] S. Ren, K. He, R. Girshick and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems* 28 (2015).

[57] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, 4510–4520.

[58] N. Sharma, V. Jain and A. Mishra. An analysis of convolutional neural networks for image classification. *Procedia computer science* 132 (2018), 377–384.

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

[60] K. Sims. Evolving virtual creatures. *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. 1994, 15–22.

[61] R. D. Singh, A. Mittal and R. K. Bhatia. 3D convolutional neural network for object recognition: a review. *Multimedia Tools and Applications* 78.12 (2019), 15951–15995.

[62] H. F. Song, G. R. Yang and X.-J. Wang. Reward-based training of recurrent neural networks for cognitive and value-based tasks. *Elife* 6 (2017), e21492.

[63] C. J. Spoerer, P. McClure and N. Kriegeskorte. Recurrent convolutional neural networks: a better model of biological object recognition. *Frontiers in psychology* 8 (2017), 1551.

[64] S. Srinivas and R. V. Babu. Data-free parameter pruning for deep neural networks. *arXiv preprint arXiv:1507.06149* (2015).

[65] S. Srinivas and R. V. Babu. Data-free parameter pruning for Deep Neural Networks. *CoRR* abs/1507.06149 (2015). arXiv: 1507.06149. URL: http://arxiv.org/abs/1507.06149.

[66] B. Šter and A. Dobnikar. Neural networks in medical diagnosis: Comparison with other methods. *International conference on engineering applications of neural networks*. 1996, 427–30.

[67] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman and Z. Zhang. Hardware for machine learning: Challenges and opportunities. *2017 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE. 2017, 1–8.

[68] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich. Going deeper with convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, 1–9.

[69] J. M. Tebelskis. *Speech recognition using neural networks*. Carnegie Mellon University, 1995.

[70] V. Vanhoucke, A. Senior and M. Z. Mao. Improving the speed of neural networks on CPUs. (2011).

[71] A. Waibel. Modular construction of time-delay neural networks for speech recognition. *Neural computation* 1.1 (1989), 39–46.

[72] J. Wen, B. Zhang, Y. Xu, J. Yang and N. Han. Adaptive weighted nonnegative low-rank representation. *Pattern Recognition* 81 (2018), 326–340.

[73] L. Wu, Y. Wang, J. Gao and X. Li. Deep adaptive feature embedding with local sample distributions for person re-identification. *Pattern Recognition* 73 (2018), 275–288.

[74] Y. Wu, Y. Wu, R. Gong, Y. Lv, K. Chen, D. Liang, X. Hu, X. Liu and J. Yan. Rotation consistent margin loss for efficient low-bit face recognition. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, 6866–6876.

[75] 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. *2016 24th European Signal Processing Conference (EUSIPCO)*. IEEE. 2016, 1862–1866.

[76] Z. Xu, Y.-C. Hsu and J. Huang. Training shallow and thin networks for acceleration via knowledge distillation with conditional adversarial networks. *arXiv preprint arXiv:1709.00513* (2017).

[77] J. Yim, D. Joo, J. Bae and J. Kim. A gift from knowledge distillation: Fast optimization, network minimization and transfer learning. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, 4133–4141.

[78] S. S. Young, P. D. Scott and N. M. Nasrabadi. Object recognition using multilayer Hopfield neural network. *IEEE Transactions on Image Processing* 6.3 (1997), 357–372.

[79] S. Zagoruyko and N. Komodakis. Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer. *arXiv preprint arXiv:1612.03928* (2016).

[80] X. Zhang, X. Zhou, M. Lin and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, 6848–6856.

[81] T. Zhao, X. He, J. Cheng and J. Hu. Bitstream: Efficient computing architecture for real-time low-power inference of binary neural networks on cpus. *Proceedings of the 26th ACM international conference on Multimedia*. 2018, 1545–1552.

[82] F. Zhu, R. Gong, F. Yu, X. Liu, Y. Wang, Z. Li, X. Yang and J. Yan. Towards unified int8 training for convolutional neural network. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, 1969–1979.

PUBLICATIONS

PUBLICATION 1

Binarized Convolutional Neural Networks for Efficient Inference on GPUs

Mir Khan, Heikki Huttunen, Jani Boutellier

2018, pp. 682-686

10.23919/EUSIPCO.2018.8553594

Binarized Convolutional Neural Networks for Efficient Inference on GPUs

Mir Khan, Heikki Huttunen, Jani Boutellier
Tampere University of Technology
Tampere, Finland
Email: Mir.Khan@tut.fi, Heikki.Huttunen@tut.fi, Jani.Boutellier@tut.fi

***Abstract*—Convolutional neural networks have recently achieved significant breakthroughs in various image classification tasks. However, they are computationally expensive, which can make their feasible implementation on embedded and low-power devices difficult. In this paper convolutional neural network binarization is implemented on GPU-based platforms for real-time inference on resource constrained devices. In binarized networks, all weights and intermediate computations between layers are quantized to +1 and -1, allowing multiplications and additions to be replaced with bit-wise operations between 32-bit words. This representation completely eliminates the need for floating point multiplications and additions and decreases both the computational load and the memory footprint compared to a full-precision network implemented in floating point, making it well-suited for resource-constrained environments. We compare the performance of our implementation with an equivalent floating point implementation on one desktop and two embedded GPU platforms. Our implementation achieves a maximum speed up of $7.4\times$ with only 4.4% loss in accuracy compared to a reference implementation.**

Keywords: model compression, binarized convolutional neural networks, optimization, image classification

1. Introduction

In the recent years, convolutional neural networks (CNNs) have presented impressive performance in image classification [16][4], face recognition [17][19], audio classification [14], and speech recognition [7]."

Large neural network models can be computationally expensive, making them unsuitable for deployment to small resource-constrained mobile devices. To this extent, contemporary CNN-based solutions often acquire the input data on a mobile device, but transmit the data to a remote server for CNN-based processing. However, performing the CNN-based processing on the mobile device (a.k.a. edge computing) would reduce the overall system complexity and enable real-time applications.

The emerging CNN subfield of *model compression* aims to retain the accuracy of the neural network while minimizing redundant network parameters and reducing computational load. Many such techniques have already been proposed.

One technique [9] is based on *pruning* of parameters, where majority of the parameters of the network are removed without significantly impacting accuracy. Reduction of parameters initially leads to a significant drop in accuracy; however, retraining (fine-tuning) of the parameters restores most of the network's accuracy. The authors report $13\times$ reduction of memory requirements with no loss in accuracy [9].

Another approach, low-rank approximation of convolutional kernels [13], approximates 2D convolutions with convolutions by vectors. The separable kernels can be obtained either by training the network with separable filters [1] or by posing it as an optimization problem to minimize the reconstruction error of the feature maps. Depending on the approach [13][1], speedups between $2\times$ to $4\times$ have been reported on CPU implementations.

Binarized neural networks (BNN) have been first introduced in [11], where their performance was demonstrated on the CIFAR-10 dataset. The weights and activations for intermediate computations are binarized to $+1$ and -1. The authors present a speed up of $7\times$ on a network for the MNIST dataset. In a further work [21] the approach was refined for CPU implementation and evaluated on the ImageNet dataset.

by packing 1-bit weights into 32-bit words, enabling replacement of multiplication operations by logic XNORs. In this paper, an approach for the implementation of BNNs [11] on GPU platforms is presented. To the best knowledge of the authors, this is the first work that presents a GPU implementation of a binarized convolutional neural network for inference. We present our implementation with an application use case of vehicle type classification [12]. Results show significant speedups in real-time inference compared to a floating point version of an equivalent neural network.

As a summary, the contributions of this work are as follows:

- Detailed presentation of efficiently implementing CNN binarization, including the convolutional layers, on GPU-based platforms.
- Comparison of different approaches for binarizing input data, and how each approach impacts the classification accuracy.
- Performance (execution time) comparisons on several platforms.

The source code for our CUDA implementation is publically available [1].

2. Experimental Setup

2.1. Binarizing the network

Our binarized network architecture is based on the original vehicle classifier network presented in [12]. We implement a binarized version of the same architecture in several steps. We do not use any ReLU [6] activations in the binarized version. In the original binarization work [5], the authors suggest two approaches for binarization: stochastic and deterministic. For binarizing the weights and intermediate computations, we use the deterministic *sign* function, which is defined as

$$sign(x) = \begin{cases} -1 & \text{if } x \leq 0 \\ +1 & \text{if } x > 0 \end{cases} \quad (1)$$

For training the BNN, following [10], we explicitly define the gradient of the *sign* function to be the identity function in the backward pass, such that $\frac{\partial sign(x)}{\partial x} = x$.

The non-binarized network is trained with the RMSprop optimizer [23], while the binarized version is trained with the ADAM [15] optimizer. After training, only the binarized weights are used for inference for the binarized network.

The network is trained with a dataset set consisting of 6555 images of vehicles that have been captured by a camera and manually categorized into four different classes: *bus*, *normal*, *truck*, and *van*. Each image has size 96×96 and are in full color. The data has been split into a training set (90%) and a test set (10%). We augment the training set using flipping and filtering with a 2D Gaussian filter with $\sigma = 0.5$, resulting in a total training set size of 14,108 images, 20% of which are used for validation. Throughout this text, our accuracy reports are for the performance of the network on the test set that corresponds to the best validation set accuracy.

2.2. Testing pipeline

For obtaining runtime results, we use the built-in GPU timers to measure the runtime of the kernels for our CUDA and OpenCL programs. Our kernel execution time measurements do not include memory transfer times to/from the GPU, as they can be affected by various factors, some of which are hardware-dependent, for example, on the NVidia Jetson host and device memory are shared. The correctness and accuracy of the profiling results generated have been verified by the Nvidia Visual Profiler for the same CUDA programs.

For each test run, 1000 images are randomly generated and fed to the network one at a time. The timer begins after the memory is copied, and the timer ends after the last kernel's computation is completed. Our final result is the total accumulated time per sample averaged over all 1000 samples.

1. github.com/Valentin4869/BinCNN

2.3. Input binarization

In this section we describe our methods for binarizing the inputs to the first layer of our BNN. We pre-process the data set using these techniques and evaluate the accuracy of the BNN on the pre-processed data set.

Thresholding A constant threshold T can be subtracted from the input $\mathbf{X}$ before binarizing it. We simply substitute the input $\mathbf{X}$ to the first layer with $sign(\mathbf{X} + T)$, for $\mathbf{X} \in \mathbb{R}^{H \times W \times C}$, and for $T \in \mathbb{R}^{1 \times 1 \times C}$. The motivation is to shift the range of values taken by $\mathbf{X}$ such that binarization with the *sign* function produces meaningful results, as opposed to all zeros for standard pixel-value ranges do not include negative numbers. The network is trained as before but in two stages: first, the network is trained for 50 epochs and the loss is minimized with respect to all network parameters except for T. Then a second stage of tuning is entered where we minimize the loss with respect to the parameter T and the validation set. We repeat this process for several thousand training epochs until the performance on the validation set no longer improves.

Figure 1. Input binarization with RGB Thresholding (first row) and LBP (second row).

Local Binary Patterns (LBP) A well-known technique called *local binary patterns* for extracting multi-resolution and scale-invariant features from images has been introduced in [18]. We use a similar approach in our application for image binarization, but with a slight modification: we operate on the grayscale image and process each pixel by examining its neighborhood at a radius of 1 pixel, generate 3 artificial color channels and select 3 pixels at a clockwise stride of 3 in the neighbourhood to distribute to these channels. Then the value of these pixels are set to 1 if they exceed the value of the center pixel and 0 otherwise. An example of this transformation on an image from the dataset is demonstrated in the second row of Figure 2.3.

2.4. Packing binary-valued vectors

To avoid confusion with terminology, we denote by *packing* the encapsulation/conversion of an array of 1-bit values into an individual 32-bit unsigned integer. Formally,

for a binary-valued vector $\mathbf{x} \in \{-1,+1\}^D$, assuming D is divisible by B, then the packed representation of $\mathbf{x}$, $\mathbf{x}_p \in \{-1,+1\}^{D/B}$ for a packing bitwidth $B \leq 32$ (assuming 32-bit word) and positive D, is given by

$$\mathbf{x}_p = \begin{bmatrix} \sum_{i=1}^{B}(1+x_i)2^{B-2-mod(i-1,B)} \\ \sum_{i=B+1}^{2B}(1+x_i)2^{B-2-mod(i-1,B)} \\ \sum_{i=2B+1}^{3B}(1+x_i)2^{B-2-mod(i-1,B)} \\ \vdots \\ \sum_{i=D-B+1}^{D}(1+x_i)2^{B-2-mod(i-1,B)} \end{bmatrix}. \quad (2)$$

3. Implementation

In this section, we present the details of our CUDA implementation of the binarized neural network architecture described in Section 2. We use CUDA terminology throughout this section.

3.1. Convolutional layers

The convolutional layer in a neural network can significantly improve image classification accuracy compared to standard multi-layer perceptrons. Given a kernel $H \in \mathbb{R}^{K\times K\times C}$ and an image $X \in \mathbb{R}^{H\times W\times C}$, an output feature map $F \in \mathbb{R}^{H\times W}$ is given by the expression

$$F[i,j] = \sum_{c=0}^{C-1}\sum_{l=-R}^{R}\sum_{k=-R}^{R} H[R+l,R+k,c]X[i+k,j+l,c], \quad (3)$$

for odd K, and the kernel radius $R = \frac{K-1}{2}$. It should be noted that equation (3) in fact computes cross-correlation (not convolution), which is the convention in deep learning. A common approach for computing convolutions efficiently is through matrix multiplication [2], where the weights and image tensors are reshaped into 2-dimensional matrices, which will then allow us to compute the convolution through a single matrix multiplication. The reshaping for the weights is trivial, and this step can often be skipped if the weights are already stored in this layout; however, the process of arranging the input image into the matrix of columns used for computing the convolution can be difficult to optimize. This is due to inefficient access patterns, complicated index calculations that involves many division and modulo operations, and the overhead of storing the large output matrix to global memory.

A straightforward approach for avoiding inefficient access patterns is to load regions from the image into shared memory (on-chip memory) and then extract the patches from shared memory [3]. For an image with dimensions $H \times W \times C$ corresponding to height, width, and channels respectively, and a $K \times K \times C$ kernel with a radius of $R = \frac{K-1}{2}$, we use threadblock dimensions of $S \times W$ ($S = 2$ in our case), which covers the entire width of the image, eliminating the need to redundantly load the horizontal non-zero halo regions which are difficult to load with an efficient access pattern. Then each thread-block loads an image region of dimensions $(S + 2R) \times W$ into a region in shared memory in three steps, starting by loading the top vertical halo region, the middle part, then the bottom vertical halo region (except when loading from the bottom of the image). The shared memory buffer is zero-initialized in order to implicitly handle horizontal zero-padding. Loading vertical halo regions can be done very efficiently since all threads in the threadblock load from contiguous regions in the image array.

In the second stage, the patches of size $K \times K \times C$ are extracted from shared memory. We avoid division and modulo operations in the patch-extraction stage by using an integer counter register. This results in a $2\times$ performance boost in our case. Since the network is binarized, the packing and patch-extraction step can be fused into one step to avoid redundant accesses to global memory, reducing global memory stores by $K \times K$. The algorithm for the combined step of extracting the patches and packing them is shown in Algorithm 1.

Algorithm 1 Patch-extraction and packing

```
1:  function ExtractPacked(sh_block):
2:      v ← 0
3:      k ← 0
4:      for i = 0 to B − 1 do
5:          if (i − kK = K) then
6:              k++
7:          idx = (W + 2R)(_t_y + k) + _t_x + i − kK
8:          s = sh_block[idx] > 0
9:          v = bitOR( v, s << B − 1 − i)
10:     return v
```

In Algorithm 1, `sh_block` is the region of the image loaded into shared memory using the previously described steps, including the halo regions. $\texttt{\_t}_x$ and $\texttt{\_t}_y$ are the thread indices for the x and y dimensions of the thread block corresponding to the CUDA `threadIdx.x` and `threadIdx.y` variables. B is the packing bitwidth, chosen to be 25 in our case, $<<$ is the left bit-shift operator, and v is the packed extracted patch.

For computing the convolution we implement a standard matrix multiplication subroutine in a manner similar to [22], where tiles from each matrix are loaded successively into shared memory and used to compute a submatrix of the output, such that each thread computes a single element in the output matrix, but instead of computing multiplications, we compute xnors and bit-counts following an approach similar to what was suggested in [11] as

$$\mathbf{a}\cdot\mathbf{b} = \mathtt{W} - 2 \times \mathtt{popcount}(\mathtt{xor}(A,B)), \quad (4)$$

where A and B are both 32-bit unsigned integer registers containing the packed representations of vectors **a**, **b** $\in \{-1,+1\}^{\mathtt{W}}$ respectively. We denote by $\cdot$ the real-valued dot product. The operation `xor` is the bit-wise xor operation, and `popcount` is a function for computing the number of bits set to 1. The packing bitwidth `W` is the number

TABLE 1. RUNTIME OF THE NETWORK ON EACH PLATFORM

Implementation Method	GTX1080	Mali T860	Tegra X2
cuDNN (full-precision)	$401.83\mu s$	N/A†	2.27 ms
Arm CL (full-precision)	N/A†	29.61 ms	N/A†
BCNN	$102.39\mu s$	23.63 ms	0.53 ms
BCNN with binarized inputs	$\mathbf{55.63}\mu s$	**17.58** ms	**0.41** ms

†Library not compatible with this platform.

TABLE 2. RUNTIME PER-LAYER (GTX1080)

Layer	cuDNN	Binarized	Speed-up
Im2col3d $(96, 96, 3)$	$21.63\ \mu s$	$3.17\mu s$	$6.82\times$
GEMM-convolution $(32, 5, 5, 3)$	$37.54\mu s$	$8.61\mu s$	$4.36\times$
Max-Pooling $(96, 96, 32)$	$5.22\mu s$	$8.26\mu s$	$0.63\times$
Im2col3d $(48, 48, 32)$	$65.41\mu s$	$5.50\mu s$	$11.89\times$
GEMM-convolution $(32, 5, 5, 32)$	$69.28\mu s$	$8.10\mu s$	$8.55\times$
Max-Pooling (48,48,32)	$5.38\mu s$	$2.66\mu s$	$2.02\times$
Fully-Connected $(100, 24 \times 24 \times 32)$	$200.03\mu s$	$6.28\mu s$	$31.85\times$

of elements that are packed together in a single unsigned integer register.

3.2. Fully connected layer

For the fully-connected layer, we follow a slightly different approach from standard matrix multiplication. For a packed weights matrix $\mathbf{W} \in \mathbb{R}^{\mathbf{L}\times\mathbf{D}}$, and a packed vector $\mathbf{x} \in \mathbb{R}^{\mathbf{D}\times\mathbf{1}}$, we divide the process of computing the dot product of each weight vector and $\mathbf{x}$ into 64 segments, such that each of 64 threads handling a weight vector compute the partial sum of the dot product between a weight vector and $\mathbf{x}$ through xnor operations, and stores the results in shared memory. The partial sums are then combined in a parallel reduction sum that does not require synchronization (for a warp size of 32 on the target platform).

4. Results

In this section we present our results for the impact of input binarization on classification accuracy and the performance improvement achieved.

Input binarization in Table 3 we report the classification accuracy results we obtained using each different input binarization scheme for our binarized version of the vehicle classifier [12]. We can observe that accuracy is best retained when the first layer is not binarized; however, only a moderate loss in accuracy occurs when using LBP and RGB Thresholding. Considering that RGB Thresholding is much simpler to implement and results in almost no additional computational overhead, we choose this approach for our final binarized architecture, for which we report the speed up results in the following section.

Performance Boost We time our binarized implementation on 3 different hardware platforms: Nvidia GTX 1080, Nvidia Jetson (Tegra X2), and the Mali-T860. We derive an OpenCL version of our implementation for testing on the Mali-T860, which is a straightforward process. We compare the performance of our implementation against an equivalent full precision version of the same network implemented with highly optimized libraries on each target platform, in our case these are cuDNN on Nvidia platforms, and the ARM Compute Library on the Mali-T860. We list in Table 1 the average execution times of the full network on each platform. We can see that our binarized implementation can achieve up to $7.5\times$ speed up on the GTX1080 and about $5.5\times$ on the Tegra X2. We also notice that the relative performance improvement on Mali GPU is much smaller at about $1.7\times$ for the fully binarized version. In our optimizations, we heavily take advantage of using local memory (in OpenCL terms) which resides on-chip in most workstation GPUs and the Nvidia Tegra X2, but this does not offer any performance benefits on Mali GPUs since local memory is allocated in global memory. It should be noted that cuDNN is optimized for batch processing and that our results are for one sample at a time which means these results may not necessarily be reflective of the full potential of cuDNN; however, batch processing is not a suitable option for real-time applications where a single input is processed at a time. Additionally, we note that for our cuDNN implementations, we use the explicit GEMM convolution algorithm, which can be slightly slower than the implicit GEMM algorithm. For example, cuDNN with implicit GEMM can run at $316\mu s$ for the first convolutional layer in our network on the GTX1080.

For a more detailed comparison, we present the execution times for each individual layer in Table 2. Each layer's name is followed with the dimensions of the input, except for the convolution layers where the dimensions are for the kernels, and the input dimensions can be inferred from the previous layer. This table compares the execution time of our binarized implementations with the full-precision versions of the same layer in cuDNN on the GTX1080. We omit from the table the computation times for ReLU activations, which are present in the full-precision version of the network, but are absent from the binarized version. We also omit the last 2 fully-connected layers since they are too small and in most practical applications it would be more efficient to implement them on the CPU. We include the computation time for packing the outputs of the previous layer in the binarized version of the fully-connected layer for a fair comparison. The results in Table 2 have been obtained directly from the Nvidia Visual Profiler.

It should be noted that the runtime for the fully-connected layer for full-precision cuDNN in Table 2 includes a matrix transposition. The run time excluding matrix transposition is about $100\mu s$; however, it is a necessary step for evaluating this layer. Our full-precision matrix multiplication kernel is in fact $2\times$ slower than cuBLAS (as measured in this network), yet a significant speed-up is still achievable through binarization.

TABLE 3. IMPACT OF DIFFERENT INPUT-BINARIZATION SCHEMES ON CLASSIFICATION ACCURACY

Method	Accuracy
LBP	92.06%
Thresholding Grayscale	89.16%
Thresholding RGB	92.52%
No input binarization	94.20%
Full-precision network	97.09%

5. Conclusion and Future Work

We presented an efficient implementation of a binarized convolutional neural network on GPUs that can achieve a significant decrease in runtime while reasonably preserving classification accuracy. In the future we wish to restructure our algorithms to achieve a similar performance improvement on other embedded platforms. We are also planning to extend this work to alternative convolution algorithms such as implicit GEMM, which can be faster than explicit GEMM. Finally, we plan to extend our study of how input binarization impacts classification accuracy on larger datasets with more difficult classification tasks.

Acknowledgment

This work was funded by the Academy of Finland project 309903 CoEfNet.

References

[1] Alvarez J, Petersson L. Decomposeme: Simplifying convnets for end-to-end learning. arXiv preprint arXiv:1606.05426. 2016 Jun 17.

[2] Chellapilla K, Puri S, Simard P. High performance convolutional neural networks for document processing. In Tenth International Workshop on Frontiers in Handwriting Recognition 2006 Oct 23.

[3] Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E. cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759. 2014 Oct 3.

[4] Ciregan D, Meier U, Schmidhuber J. Multi-column deep neural networks for image classification. In Computer Vision and Pattern Recognition (CVPR), IEEE Conference on 2012.

[5] Courbariaux M, Bengio Y, David JP. Binaryconnect: Training deep neural networks with binary weights during propagations. In Advances in neural information processing systems 2015 (pp. 3123-3131).

[6] Glorot X, Bordes A, Bengio Y. Deep Sparse Rectifier Neural Networks. In Aistats 2011 (Vol. 15, No. 106, p. 275).

[7] Graves A, Jaitly N. Towards End-To-End Speech Recognition with Recurrent Neural Networks. In ICML 2014 (Vol. 14, pp. 1764-1772).

[8] Gregor K, Danihelka I, Graves A, Rezende DJ, Wierstra D. DRAW: A recurrent neural network for image generation. arXiv preprint arXiv:1502.04623. 2015.

[9] Han S, Pool J, Tran J, Dally W. Learning both weights and connections for efficient neural network. InAdvances in neural information processing systems 2015 (pp. 1135-1143).

[10] Hinton, G. Neural networks for machine learning. Coursera, video lectures. 2012.

[11] Hubara I, Courbariaux M, Soudry D, El-Yaniv R, Bengio Y. Binarized neural networks. In Advances in neural information processing systems 2016 (pp. 4107-4115).

[12] Huttunen H, Yancheshmeh FS, Chen K. Car type recognition with deep neural networks. In Intelligent Vehicles Symposium (IV), IEEE 2016 Jun 19 (pp. 1115-1120).

[13] Jaderberg M, Vedaldi A, Zisserman A. Speeding up convolutional neural networks with low rank expansions. arXiv preprint arXiv:1405.3866. 2014 May 15.

[14] Kanda N, Takeda R, Obuchi Y. Elastic spectral distortion for low resource speech recognition with deep neural networks. In Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on 2013 (pp. 309-314).

[15] Kingma DP, Ba J. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980. 2014 Dec 22.

[16] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems 2012 (pp. 1097-1105).

[17] Lawrence S, Giles CL, Tsoi AC, Back AD. Face recognition: A convolutional neural-network approach. IEEE transactions on neural networks. 1997;8(1):98-113.

[18] Ojala T, Pietikäinen M, Mäenpää T. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. IEEE Transactions on pattern analysis and machine intelligence. 2002 Jul; 24(7):971-87.

[19] Parkhi OM, Vedaldi A, Zisserman A. Deep Face Recognition. In BMVC 2015 (Vol. 1, No. 3, p. 6).

[20] Pedersoli F, Tzanetakis G, Tagliasacchi A. Espresso: Efficient Forward Propagation for BCNNs, ICLR 2018 (to appear).

[21] Rastegari M, Ordonez V, Redmon J, Farhadi A. Xnor-net: Imagenet classification using binary convolutional neural networks. In European Conference on Computer Vision 2016 Oct 8 (pp. 525-542).

[22] Tan G, Li L, Triechle S, Phillips E, Bao Y, Sun N. Fast implementation of DGEMM on Fermi GPU. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis 2011 Nov 12 (p. 35).

[23] Tieleman T, Hinton G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning. 2012.

PUBLICATION 2

Comparing Optimization Methods of Neural Networks for Real-time Inference

Mir Khan; Henri Lunnikivi; Heikki Huttunen; Jani Boutellier

2019, pp. 1-5,

doi: 10.23919/EUSIPCO.2019.8902760.

Comparing Optimization Methods of Neural Networks for Real-time Inference

Mir Khan*, Henri Lunnikivi*, Heikki Huttunen, Jani Boutellier
University of Tampere
Tampere, Finland
mir.markhan@tuni.fi, henri.lunnikivi@tuni.fi, heikki.huttunen@tuni.fi, jani.boutellier@tuni.fi
*Indicates equal contribution

***Abstract*—This paper compares three different optimization approaches for accelerating the inference of convolutional neural networks (CNNs). We compare the techniques of separable convolution, weight pruning, and binarization. Each method is implemented and empirically compared in three aspects: preservation of accuracy, storage requirements, and achieved speed-up. Experiments are performed both on a desktop computer and on a mobile platform using a CNN model for vehicle type classification. Our experiments show that the largest speed-up is achieved by binarization, whereas pruning achieves the largest reduction in storage requirements. Both of these approaches largely preserve the accuracy of the original network.**

Keywords: convolutional neural networks, model optimization, image classification

1. Introduction

Convolutional neural networks (CNNs) have demonstrated notable performance in a range of practical tasks, including image classification [14][3], face recognition [15][17], audio classification [11], and speech recognition [5]. At the same time as CNNs have become significantly popular, their deployment to small-scale devices has been hindered by the considerable memory- and computation time requirements of CNN models. To address this issue, several approaches [10][8][4][6] for CNN *model optimization* have been developed in the recent years.

Some techniques such as *weight pruning* [6] aim to reduce the number of weights in the model, which in turn reduces memory requirements for running and storing the model. It has also been shown that a pruned network with sufficient sparsity allows the computations to be performed through efficient procedures for handling sparse matrices, such as sparse matrix-vector multiplication [2]. Studies (e.g. [6]) have shown that a considerable portion of network weights can be removed without significantly impacting accuracy: the authors of [6] report $13\times$ reduction of memory requirements with no loss in accuracy.

The *separable convolution* (SepConv) approach introduced in [10] approximates 2D convolutions with convolutions by vectors, i.e. separable rank-1 kernels. The separable kernels can be obtained either by training the network from scratch with separable filters [1] or by formulating the process as an optimization problem where the reconstruction error of the feature maps using separable filters is minimized. These approaches [10][1], report speedups between $2\times$ to $4\times$ on CPU implementations.

Neural network *binarization* is a CNN quantization technique that has been introduced in [8]. The principal idea of binarization is to quantize weights, activations and intermediate computations to values of $+1$ and -1. The authors of [8] show a speedup of up to $7\times$ on a network for classifiying the MNIST dataset. Later works [18] extended the binarization to larger datasets such as ImageNet.

The SepConv and weight pruning approaches are similar in the sense that both approximate and optimize the CNN model by *reducing* the number of parameters, while keeping the precision of the remaining weights the same as in the original model; in general as 32-bit floats. In contrast, the CNN binarization optimization keeps the *number* of weights the same as in the original model, but reduces the precision of each weight. Based on existing works, it is not clear how these two alternative approaches compare in terms of preserving model accuracy, improving computation time, and reducing memory space requirements.

This paper follows optimization approaches based on separable convolution [10], binarization [8][4] and weight pruning [6], and applies them to the practical use case of CNN-based vehicle type recognition [9]. In case of the binarization optimization, our results mostly build on our previous work [12]. The different optimization approaches are compared in terms of performance, storage requirements, amount of computations and the network's accuracy compared to the uncompressed baseline.

The contributions of this work are summarized as follows:

- Detailed empirical analysis and comparison of three different model optimization methods for compression and speeding up of neural networks, namely separable convolution, weight pruning, and binarization.
- Open source optimized implementations of each method for efficient real-time inference.

Our results show that pruning offers the highest reduction in storage requirements, while the lowest execution time is attained with a binarized convolutional neural network

Figure 1. From left to right, a 'normal car', 'bus', 'truck', and a 'van'.

(BCNN). The source code for our implementations are publicly available [1][2].

2. Network Model

Our basic neural network model is that of the vehicle classifier network presented in [12], with two convolutional layers, each with 32 output feature maps, and filter dimensions 5×5. Each convolutional layer is followed by a 2×2 maxpool operation. The convolutional layers are followed by three fully connected layers with shapes 24×24×32×100, 100×100, and 100×4. Each optimization approach we are using slightly alters the structure of the network. This will be discussed in Section 3.

The dataset used for training the network consists of 6555 photos of vehicles captured by a camera and manually categorized into four categories: *bus*, *normalcar*, *truck*, and *van*. Each image is of size 96×96 and is in full color. The data has been split into a training set (80%), validation set (10%) and a test set (10%). We use the accuracy recorded on the test set that corresponds to the best validation set accuracy as our final accuracy report. Figure 1 shows an example image from each class.

We implement all of our training procedures, including the custom model optimization approaches, in TensorFlow. For inference performance measurements, we have our own implementations for each model optimization approach. The GPU implementations are done in CUDA or OpenCL, depending on the platform tested.

3. Optimization Methods

In this section, we discuss in detail in each subsection the different optimization approaches we use and how they are incorporated to our network model. Each subsection begins by introducing the algorithmic changes, and then proceeds to explain how the method is implemented for inference purposes.

3.1. Separable Convolution

Convolutional layers in a CNN can be replaced by spatially separable convolution layers to reduce the number of computations and the number of parameters [10]. Essentially, a separable convolution layer will approximate a standard convolutional layer as follows

$$\sum_{c=1}^{C} Z_c^f * x_c \approx \sum_{k=1}^{K} h_k^f * \sum_{c=1}^{C} v_c^k * x_c \tag{1}$$

In Equation 1, $*$ denotes the convolution operation. The convolution of the image $x \in \mathbb{R}^{W \times H \times C}$ with each $Z^f \in \mathbb{R}^{5 \times 5 \times C}$, which is full rank, is approximated by two convolutions with $h^f \in \mathbb{R}^{1 \times 5 \times K}$, and $v^k \in \mathbb{R}^{5 \times 1 \times C}$. C is the number of channels (usually three), and K denotes the number of intermediate feature maps, which are computed by convolving the image by each of v^k (K in total), which are then in turn convolved by each of h^f kernels (F in total, corresponding to the original number generated by the full rank kernel Z^f). The result in Equation 1 is the computation of the fth feature map. The choice of number of intermediate feature maps K has an impact on the number of computations, memory requirements, and the network's accuracy. Therefore, it is imperative to carefully select a value for K that offers the best trade off over all these factors.

In our application, we choose the smallest value for K that achieves the best accuracy. We discuss this in more detail in Section 4. The separable filters can either be obtained by training the network with separable filters [1] or by minimizing the reconstruction error of the outputs of each convolution layer [10], which allows us to state the loss to be minimized with respect to the separable filters as follows (for the fth set of filters in the lth layer):

$$\mathcal{L}(\mathbf{X}) = \frac{1}{N} \sum_{x \in \mathbf{X}} \Big\| \sum_{c=1}^{C} Z_c^f * \Psi_{l-1}(x) - \sum_{k=1}^{K} h_k^f * \sum_{c=1}^{C} v_c^k * \Psi_{l-1}(x) \Big\|_2^2, \tag{2}$$

where the output of lth layer in the network denoted by $\Psi_l(x)$ on input x, so that $\Psi_0(x) = x$, and Ψ_1 is the output of the first convolutional layer (or alternatively, the input to the 2nd layer). The loss is then averaged across the entire set $\mathbf{X}$ of N training samples.

We optimize with respect to each set of separable filters independently for each convolutional layer, resulting in four different optimization steps for each epoch for the entire network. We use the RMSProp [19] optimizer with a learning rate 0.001 and decay rate 0.95. We compare the accuracy of the separable network with the original filters throughout the optimization process and stop the optimization when the validation accuracy stops improving (about 1000 epochs in our case).

Table 1 depicts the development of SepConv accuracy as a function of intermediate feature maps K. It can be seen that the highest accuracy is already reached at $K = 7$. Considering the nearly linear increase in execution time, $K = 7$ becomes the value of choice in our later experiments and comparisons with other optimization methods.

1. github.com/Valentin4869/vcoptimizations
2. github.com/hegza/vcn-inference-rs

TABLE 1. INFERENCE CHARACTERISTICS OF SEPARABLE CONVOLUTION LAYER AS A FUNCTION OF FEATURE MAP COUNT K

K	Accuracy	Performance	Memory (KB)
1	87.3%	45 μs	1.9336
2	95.1%	48 μs	3.8672
3	96.6%	53 μs	5.8008
4	96.6%	58 μs	7.7344
5	97.0%	64 μs	9.668
6	97.1%	69 μs	11.6016
7	97.2%	75 μs	13.5352
8	97.1%	80 μs	15.4688
9	97.1%	85 μs	17.4023
10	97.1%	92 μs	19.3359
11	97.1%	97 μs	21.2695
12	97.3%	102 μs	23.2031
13	97.1%	109 μs	25.1367
14	97.3%	115 μs	27.0703
15	97.3%	121 μs	29.0039
16	97.4%	127 μs	30.9375

3.2. Pruning

Pruning is a method for reducing the number of weights in a network in the fully-connected layers [6], resulting in a sparse weights matrix, which allows the computation to be performed more efficiently using sparse matrix-vector multiplication [2]. We prune the network parameters in the lth layer that fall below the threshold $T = \frac{max(W^l)+min(W^l)}{2}$ by fixing them to 0. This initially results in a significant drop in accuracy compared to the baseline model. The network is then retrained for several epochs, which restores the accuracy of the network. A variant of this method [6] uses L2 regularization during the network training, which further sparsifies the layer weights.

We use the L2-regularized pruning approach where we first train the network normally, then we load the weights into a new model that is pruned and fine-tuned in two stages: first, the network is retrained for several (e.g. 30) epochs, where all the parameters are updated, using L2 regularization. The stage that follows removes all weights that are below T and then retrains (or fine-tunes) all the network parameters except for the ones removed in the pruning stage. This process is illustrated in Algorithm 1. We use a value of $\lambda = 0.35$ in our application for the L2-regularized loss function.

For inference, we implement a sparse matrix-vector multiplication procedure in a manner similar to [2]. The sparse weights matrix is reordered to the Compressed Sparse Row (CSR) format, which then allows the computation to be performed efficiently by eliminating all weights that have been pruned from the matrix.

3.3. Weights Quantization

Binarization is an optimization approach that reduces the precision of network weights and activations to 1-bit. The concept was first introduced and demonstrated in [8]. We follow this approach and replace all ReLU activations with the sign function defined as

$$sign(x) = \begin{cases} -1 & \text{if } x \leq 0 \\ +1 & \text{if } x > 0 \end{cases} \tag{3}$$

Additionally, we binarize all the weights in the network in the same way and use an approach identical to `vcbcnn` [12] to implement the binarized version of the vehicle classification network for inference.

For training the BCNN, following [7], we explicitly define the gradient of the *sign* function to be the identity function, such that $\frac{\partial sign(x)}{\partial x} = x$. Gradients are in full precision and are not binarized and the updates during training are done to the full-precision weights.

The binarized version is trained with the ADAM [13] optimizer. After training, only the binarized weights are used for inference. Binary-valued weights can be 'packed' such that each subgroup of 32 binary weights can occupy whole 32-bit registers, allowing for a reduction in storage size of the network parameters by up to 32×. This also allows for computing dot products much more efficiently as follows:

$$\mathbf{a} \cdot \mathbf{b} = \mathtt{W} - 2 \times \mathtt{popcount}(\mathtt{xor}(A, B)), \tag{4}$$

In Equation (4), both A and B are 32-bit unsigned integers containing the packed (decimal) representations of $\mathbf{a}$, $\mathbf{b} \in \{-1, +1\}^{\mathtt{W}}$, i.e., converted from an array of 32 bits to a 32-bit unsigned integer. The real-valued dot product is denoted by the $\cdot$ operator. The function `xor` is the bit-wise xor operation, and `popcount` is a function for computing the number of bits set to 1. The packing bitwidth `W` denotes the number of elements (or bits) that are packed together in a single unsigned integer register, which is 32 in our case.

Algorithm 1 Fully-connected layer pruning procedure

for l in Layers **do**
  for w in W^l **do**
    $w \leftarrow$ LoadPretrained(w)
for stage in Stages **do**
  // Retrain with L2 regularization
  for epoch in Train_Epochs **do**
    for l in Layers **do**
      for w in W^l **do**
        $w \leftarrow$ Update(w,η,$\frac{\partial \mathcal{L}(\mathbf{X})}{\partial w}$)
  // Pruning stage
  for l in DenseLayers **do**
    $T \leftarrow \frac{max(W^l)+min(W^l)}{2}$
    for w in W^l **do**
      if $w < T$ **then**
        $w \leftarrow 0$
  // Fine-tune parameters
  for epoch in Tuning_Epochs **do**
    for l in Layers **do**
      for w in NonZero(W^l) **do**
        $w \leftarrow$ Update(w,η,$\frac{\partial \mathcal{L}(\mathbf{X})}{\partial w}$)

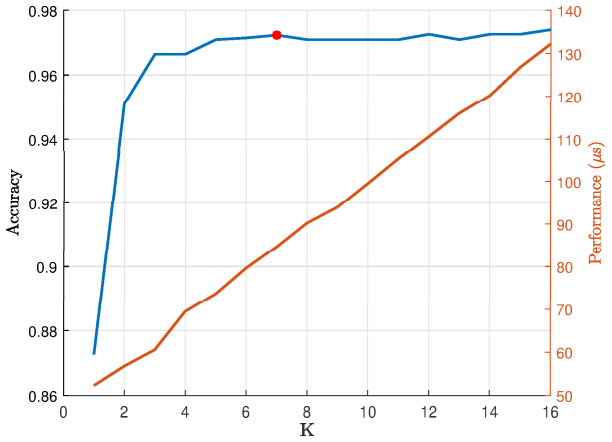


Figure 2. Performance/accuracy trade-off for different values of K.

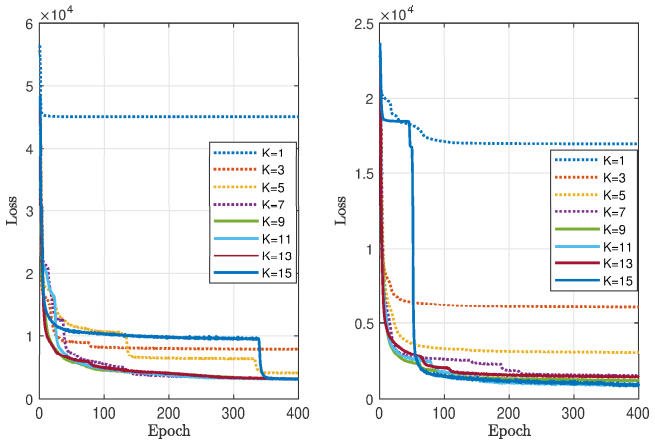


Figure 3. Loss for convolutional layers 1 (left) and 2 (right) shown for different values of K.

4. Results

This section presents the results of our experiments and compares their impacts on accuracy, performance, and memory usage for each optimization approach implemented. The two platforms used to perform the computation time measurements are provided in Table 2. From here on we refer to the platforms using their tag (GTX or Mali).

Table 1 shows the performance of convolution layers in the separable convolution network at different values of intermediate feature maps K. Performance is measured as execution time in microseconds on the GTX platform.

The graph in Figure 2 shows how the accuracy of the network changes as we change the number of intermediate featuremaps computed by the separable convolution layers. The performance of the convolutional layers for the whole network at each value of K is also shown in μs on the GTX platform. We find that $K = 7$ offers a good trade-off between accuracy and performance. While from Figure 2 it appears that for $K = 7$ the convergence is slower, the loss (Figure 3) is eventually reduced to a level comparable to other values above 7. The slow-down in convergence is not significant in our case, but it may impact larger networks and in cases where an optimizer other than RMSProp is used.

The plot of Figure 4 shows the effect of weight pruning to fully-connected layers. As the sparsity increases in the fully-connected layer, the network's accuracy begins to fluctuate at wider ranges; however, a high accuracy of 97% (marked in red) is reached at a sparsity level of nearly 99.9%. The fine-tuning stage, where pruned weights are held at 0 and are not updated, is shown in green. The blue plot marks the stage where all the weights are updated and the previously pruned weights in the previous step are restored and re-updated. All accuracy reports in Figure 4 are on the validation set.

A compact summary of these results is shown in Table 3. *Baseline (vnd. lib.)* refers to our implementation of the network using vendor-provided optimized libraries, which are cuDNN for GTX and Arm Compute Library for Mali.

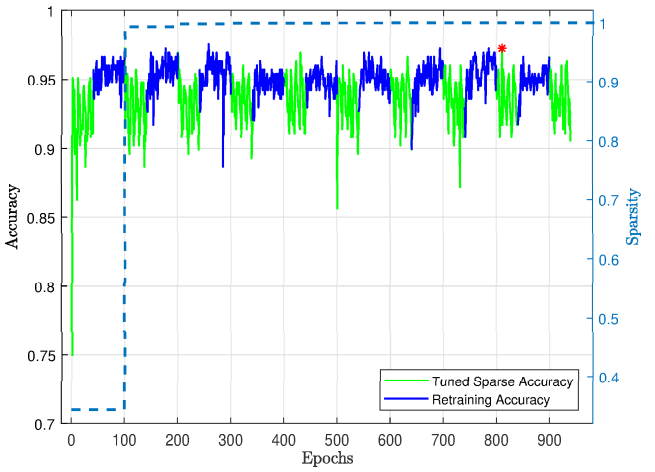


Figure 4. Accuracy of the network on the validation set at retraining stages and fine-tuning stages.

The column *Baseline (own)* is for our own low-level implementation without acceleration libraries.

We find that the best execution time is achieved through BCNN quantization on both platforms, but this approach also has the largest drop in accuracy across all methods. In contrast, on very low-resource devices (e.g. Internet of Things appliances), reduction of memory requirements can be more important than computation speed, and for this pruning outperforms BCNN by a clear margin. Combining pruning and separable convolution has the potential to further reduce the memory requirements, but the accuracy impact is not predictable and should be investigated.

5. Conclusion and Future Work

We compared several different CNN model optimization approaches for the application of vehicle classification. Our results show that significant reductions in memory requirements and computations can be achieved, however the compression technique must be selected based on the

TABLE 2. PLATFORMS USED FOR EXPERIMENTS

Tag	CPU	GPU	Operating System
GTX	Intel i7-7700K	NVidia GeForce GTX1080	Ubuntu 16.04
Mali	ARM Cortex-A72×2 + Cortex-A53×4	ARM Mali T860	Linux Firefly 4.4

TABLE 3. RESULTS OF OPTIMIZATION METHODS AT INFERENCE

	Baseline (vnd.lib.)	Baseline (own)	Pruning	SepConv	BCNN
GTX	0.4 ms	3.1 ms	0.7 ms	1.1 ms	**0.06** ms
Mali	30 ms	120 ms	66 ms	80 ms	**17.6** ms
Accuracy	97.5%	97.5%	95.5%	97.3%	94%
Memory (KB)	7350	7350	**150**	7254	230
Muls (in 10^6)	82.95	82.95	81.53	18.30	**0.43**
Adds (in 10^6)	86.12	86.12	84.35	20.37	**0.43**

optimization objective: BCNN quantization provides the highest performance, whereas pruning provides the highest reduction in storage requirements (assuming a network with significant fully-connected layers). In future works we plan to extend this study of optimization approaches to larger and more detailed datasets with real-life applications. Also, extending the analysis towards combinations of different compression methods in the spirit of [16] is definitely a direction worth investigating.

Acknowledgment

This work was funded by the Academy of Finland project 309903 CoEfNet.

References

[1] Alvarez J, Petersson L. DecomposeMe: Simplifying ConvNets for end-to-end learning. arXiv preprint arXiv:1606.05426, 2016 Jun 17.

[2] Bulu A, Fineman JT, Frigo M, Gilbert JR, Leiserson CE. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In Annual Symposium on Parallelism in Algorithms and Architectures (SPAA), 2009 (pp. 233-244).

[3] Ciregan D, Meier U, Schmidhuber J. Multi-column deep neural networks for image classification. In IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2012 (pp. 3642-3649).

[4] Courbariaux M, Bengio Y, David JP. BinaryConnect: Training deep neural networks with binary weights during propagations. In Advances in Neural Information Processing Systems (NIPS), 2015 (pp. 3123-3131).

[5] Graves A, Jaitly N. Towards end-to-end speech recognition with recurrent neural networks. In International Conference on Machine Learning (ICML), 2014 (pp. 1764-1772).

[6] Han S, Pool J, Tran J, Dally W. Learning both weights and connections for efficient neural networks. In Neural Information Processing Systems (NIPS), 2015 (pp. 1135-1143).

[7] Hinton, G. Neural networks for machine learning. Coursera, video lectures, 2012.

[8] Hubara I, Courbariaux M, Soudry D, El-Yaniv R, Bengio Y. Binarized neural networks. In Advances in Neural Information Processing Systems (NIPS), 2016 (pp. 4107-4115).

[9] Huttunen H, Yancheshmeh FS, Chen K. Car type recognition with deep neural networks. In Intelligent Vehicles Symposium (IV), 2016 (pp. 1115-1120).

[10] Jaderberg M, Vedaldi A, Zisserman A. Speeding up convolutional neural networks with low rank expansions. In British Machine Vision Conference (BMVC), 2014.

[11] Kanda N, Takeda R, Obuchi Y. Elastic spectral distortion for low resource speech recognition with deep neural networks. In IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU), 2013 (pp. 309-314).

[12] Khan M, Huttunen H, Boutellier J. Binarized convolutional neural networks for efficient inference on GPUs. In European Signal Processing Conference (EUSIPCO), 2018 (pp. 682-686).

[13] Kingma DP, Ba J. Adam: A method for stochastic optimization. In International Conference on Learning Representations (ICLR), 2015.

[14] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems (NIPS), 2012 (pp. 1097-1105).

[15] Lawrence S, Giles CL, Tsoi AC, Back AD. Face recognition: A convolutional neural-network approach. IEEE Transactions on Neural Networks, 1997, 8(1) (pp. 98-113).

[16] Lin JH, Xing T, Zhao R, Zhang Z, Srivastava M, Tu Z, Gupta RK. Binarized convolutional neural networks with separable filters for efficient hardware acceleration. In Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2017 (pp. 27-35).

[17] Parkhi OM, Vedaldi A, Zisserman A. Deep face recognition. In British Machine Vision Conference (BMVC), 2015.

[18] Rastegari M, Ordonez V, Redmon J, Farhadi A. Xnor-net: Imagenet classification using binary convolutional neural networks. In European Conference on Computer Vision (ECCV), 2016 (pp. 525-542).

[19] Tieleman T, Hinton G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. Coursera, video lectures, 2012.

PUBLICATION 3

Neural Network-based Vehicle Image Classification for IoT Devices

Saman Payvar; Mir Khan; Rafael Stahl; Daniel Mueller-Gritschneder, Jani Boutellier

2019, pp. 148-153,

doi: 10.1109/SiPS47522.2019.9020464

Neural Network-based Vehicle Image Classification for IoT Devices

Saman Payvar
Unit of Computing Sciences
Tampere University
Tampere, Finland
saman.payvar@tuni.fi

Mir Khan
Unit of Computing Sciences
Tampere University
Tampere, Finland
mir.markhan@tuni.fi

Rafael Stahl
Chair of Electronic Design Automation
Technical University of Munich
Munich, Germany
r.stahl@tum.de

Daniel Mueller-Gritschneder
Chair of Electronic Design Automation
Technical University of Munich
Munich, Germany
daniel.mueller@tum.de

Jani Boutellier
Tampere University /
University of Vaasa, Finland
jani.boutellier@tuni.fi

***Abstract*—Convolutional Neural Networks (CNNs) have previously provided unforeseen results in automatic image analysis and interpretation, an area which has numerous applications in both consumer electronics and industry. However, the signal processing related to CNNs is computationally very demanding, which has prohibited their use in the smallest embedded computing platforms, to which many Internet of Things (IoT) devices belong. Fortunately, in the recent years researchers have developed many approaches for optimizing the performance and for shrinking the memory footprint of CNNs. This paper presents a neural-network-based image classifier that has been trained to classify vehicle images into four different classes. The neural network is optimized by a technique called binarization, and the resulting binarized network is placed to an IoT-class processor core for execution. Binarization reduces the memory footprint of the CNN by around 95% and increases performance by more than 6×. Furthermore, we show that by utilizing a custom instruction 'popcount' of the processor, the performance of the binarized vehicle classifier can still be increased by more than 2×, making the CNN-based image classifier suitable for the smallest embedded processors.**

***Index Terms*—model compression, convolutional neural networks, image classification, internet-of-things**

I. INTRODUCTION

Convolutional neural networks (CNNs) have enabled a significant advance in automatic image analysis, such as image classification [1], image segmentation [2], image captioning [3] and object detection [4]. Unfortunately, up to recently the computational requirements of CNNs have restricted their use to server or desktop class computers, although their deployment to *edge devices* could open up a variety of new applications [5]. In the Internet-of-Things (IoT), the network edge refers to devices that are within immediate connection to sensors that provide input data for the whole IoT system. Such an edge device can be a smartphone [6], or a tiny sensor node commonly equipped with less than a megabyte of RAM [7].

A CNN consists of a sequence of *layers*, of which the most common types are *fully-connected layers* and *convolutional layers*. Once a CNN has been trained [8], e.g. for image classification, the *parameters* and *weights* of the layers are fixed for deployment to a target device. On the target device, the process that evaluates given input data is called *inference*, where the input data flows through the layers of the CNN, providing the requested output (e.g. classification result) from the last layer.

In terms of computation, convolutional layers consist of repeated 2D convolutions, where the input data of the layer is convolved by 2D kernels with common sizes of 5×5, 3×3 or 1×1 [9]. The computational effort of convolutional layers grows rapidly as the size of input images or kernels grows [10]. However, it has been well-known for some time that 2D convolution can also be interpreted and computed as a 2D matrix multiplication [11]. The inference of a fully-connected layer is also commonly performed by 2D matrix multiplication.

Optimization of CNN processing can be performed by optimizing software, hardware, or both [12]. Examples for software-based optimizations are model compression [9][13] or reduction of arithmetic precision [14][12]. Software-based optimizations that target convolutional layers include separable convolution [15] and depthwise convolution [16], whereas fully-connected layers can be optimized by weight pruning [13]. All of these optimizations have some negative impact on the CNN accuracy.

Reduction of arithmetic precision, on the other hand, is not limited to separate types of layers, but can be applied to the whole CNN. Arithmetic precision can be reduced from floating-point to, e.g., 16-bit fixed point [12] with minimal degradation of CNN (classification) accuracy, or by extreme quantization down to two [17] bits or one bit [18][14] of weight precision. When the precision of weights (and possibly also input data) is reduced to a single bit, the CNN is *binarized*. Binarization dramatically reduces the memory footprint of a CNN, as the original weights, which are normally expressed in 32-bit floating point, can be represented with a single bit. This evidently has an impact on the CNN's accuracy [18]. However, besides shrinking the size of the

TABLE I
RELATED NEURAL NETWORK OPTIMIZATION WORKS

Work	Type	Optimization	Platform
Courbariaux et al. [18]	SW only	Binarization (fc layers only)	NVidia GPU
Rastegari et al. [24]	SW only	Binarization (conv and fc layers)	64-bit CPU
Khan et al. [14]	SW only	Binarization (conv and fc layers)	NVidia and OpenCL GPUs
ESPRESSO [25]	SW only	Binarization (conv and fc layers)	NVidia GPU, CPU
Park et al. [26]	HW SW	Zero skipping, Data reuse (conv layers only)	Nvidia GPU, GPU simulation
Conti et al. [27]	HW SW	Binarization (conv and fc layers)	HW accelerator for MCUs
Proposed	HW SW	Binarization (conv and fc layers)	RISC-V MCU (simulation)

network, binarization also enables CNN inference on devices that have no support for floating-point arithmetic, such as microcontrollers and FPGAs [19].

This paper presents a CNN for vehicle image classification [20] that has been binarized including the weights of all layers, as well as the input data, following the principles of our recent work [14]. However, unlike our recent work that concentrated on CNN inference on graphics processing units, in this paper we focus on microcontroller-class devices that can be found on edge nodes of an IoT system. As the target microcontroller, we have selected PULPino [21], which is based on the open-source instruction-set architecture RISC-V [22], which is gaining interest in both academia and industry.

The contributions of this paper are as follows:

- Performance and memory footprint measurements of our binarized CNN-based image classifier on a RISC-V microcontroller, and
- Optimization of binarized CNN computations by the custom instruction 'popcount' found in a proposal for RISC-V instruction set extensions [23].

The structure of this paper is as follows: Section II introduces other works related to optimization of CNNs; Section III describes the PULPino microcontroller that we use as the target device for our image classifier; Section IV covers the structure and binarization process of our CNN; Section V presents our experimental results, and Section VI concludes the paper.

II. RELATED WORK

This section describes previous works related to acceleration of CNNs, some also considering acceleration by hardware. Table I presents a summary of these works and the target platforms they consider.

Binarized neural networks (BNN) were originally introduced in [18]: network weights and activations are restricted to $+1$ and -1, which enables replacing multiplications and additions with bit-wise operations. Experiments have been performed on MNIST and CIFAR-10 datasets. The authors demonstrate a speedup of $7\times$ for a multi-layer perceptron network trained for MNIST handwritten digit classification. Experimental results are limited to GPU acceleration of binarized fully-connected layers.

Somewhat later the binarization optimization was extended to the large-scale ImageNet image classification challenge [24]. The authors of [24] concentrate on CPU targets and report up to $58\times$ execution time reduction on 64-bit CPUs for binarized convolution and fully-connected layers. Also, the authors claim an accuracy improvement of 16% compared to [18] in the ImageNet top-1 classification challenge.

Our previous work [14] was among the first ones to present GPU acceleration of both binarized convolution and fully-connected layers. Experimental results are presented for two mobile GPUs (NVidia Jetson and ARM Mali-T860), as well as for a desktop GPU (NVidia GTX1080). Layer implementations have been written from scratch in OpenCL and CUDA and made available open source. Additionally, the accuracy impact of various input image binarization approaches are analyzed.

In [25] a self-contained library ESPRESSO for binarized neural networks is presented. The library provides layer implementations in C and CUDA for both CPU and NVidia GPU targets. ESPRESSO [25] uses an optimization called *unrolling* (similar to *im2col* used in our previous work [14] and the proposed work) for reshaping tensors prior to computing convolution.

Optimization of CNN convolution operations is studied in [26]. The authors have observed that Winograd convolutions can involve a high number of multiplications by zero, especially if weight pruning (see, e.g. [13]) has been applied. This redundancy is avoided by skipping zero weights by a software-only and by a hardware-assisted approach. Additionally, the authors present a data reuse approach for reducing the number of additions. Both optimizations target NVidia GPUs.

In [27] the XNOR Neural Engine (XNE) is presented, a hardware accelerator for binary neural networks to be closely coupled with an MCU (micro controller unit) system. The XNE is capable of executing both binarized convolutional and fully-connected layers. The authors provide post-layout results where the accelerator has been placed on the same chip and same clock domain with a RISC-V microcontroller that acts as the host processor for the accelerator.

The proposed work is similar to the work of Conti et al. [27] in the sense that both consider an IoT edge computing scenario, build on binarized CNNs, and consider RISC-V MCU cores. However, a substantial difference is that the XNE accelerator of [27] is a dedicated datapath for CNNs next to the MCU core, whereas our proposed solution builds on a basic microcontroller architecture with just one custom processor instruction ('popcount') for accelerating BNNs. Evidently, the specialized circuit of [27] can achieve much higher energy efficiency than our proposed solution, whereas our solution only requires a tiny modification to a basic RISC-V MCU system, and otherwise remains very generic and capable of accelerating other types of applications as well.

Fig. 1. From left to right, a 'bus', 'normal car', 'truck', and a 'van'.

III. THE PULPINO RISC-V PROCESSOR FOR IOT APPLICATIONS

RISC-V is an open source instruction set architecture (ISA) that is gaining interest in both academia and industry [22]. The ISA is open and standardized, such that it is free to use for both academia and industry. To promote adoption of the new ISA, another goal was to design a modern ISA: it is designed in a modular way by providing a small base instruction set with optional extensions. Additionally, certain instruction opcodes are reserved for custom extensions. This flexibility allows to design RISC-V processors that are customized for special workloads, which makes the ISA interesting for specialized IoT devices.

While the open standard is just referring to the ISA itself and not any micro-architecture, the community around RISC-V has provided many open-source cores. An important motivation for open hardware is security, especially with recent micro-architecture bugs Spectre and Meltdown appearing in popular media [28][29]. Kerckhoff's principle and a long history of research suggests that open systems provide certain advantages over closed systems in terms of security [30][31][32].

The Parallel Ultra-Low-Power (PULP) project has developed several RISC-V-based microcontrollers that are suitable for IoT applications [21]. The PULPino is particularly suited for low cost, low power tasks, because it is a simple in-order single-core microcontroller with many configuration options. Due to these advantages, the custom processor used in this work was derived from the PULPino-based SoC (System-on-Chip).

IV. NEURAL NETWORK DESIGN

A. Network for Vehicle Classification

The neural network model we use is that of the vehicle classifier network presented in [20]. The network has five layers in total, starting with two convolutional layers, each one with 32 output feature maps, and kernel sizes 5×5. Each of the convolutional layers is followed by a 2×2 maxpooling operation. The second convolutional layer is followed by three fully-connected layers. The first fully-connected layer (the 3rd layer in the network) has 100 neurons, resulting in the shape 24×24×32×100. The two layers that follow have shapes 100×100, and 100×4, in that order.

The dataset we use for training the network has 6555 photos of vehicles from four categories: *bus*, *normalcar*, *truck*, and *van*. Each vehicle image is a full-color image of size 96×96. Example images from each class in the data set are shown in Fig. 1. We split the data into a training set (80%), validation set (10%) and a test set (10%). Our test-set accuracy reports are the recorded accuracy reports that correspond to the best validation set accuracy.

B. Neural Network Binarization

We implement a binarized version of the vehicle classifier network introduced in [20] reducing the precision of CNN weights and their activations to 1-bit. This concept was first introduced in [18], with reports of substantial reductions of model execution time and size. In this work, we replace all ReLU activations in the network with the sign function, which is given as

$$sign(x) = \begin{cases} -1 & \text{if } x \leq 0 \\ +1 & \text{if } x > 0 \end{cases} \tag{1}$$

We binarize the weights of the network using the $sign$ function as well. During training, the gradient of *sign* activations are explicitly defined to be the identity function in the backward pass so that $\frac{\partial sign(x)}{\partial x} = x$. The full-precision version of the network (non-binarized) is trained using the RMSprop optimizer, and the binarized version is trained with the ADAM optimizer. For the binarized version of the network, only the binarized weights, where all have a value of either -1 or $+1$, are used for inference on the target device. The network is trained from scratch using binarization in a separate training process. It would also have been possible to quantize the network to ternary values [17] (or even higher 8- or 16-bit precision), but that would have multiplied the memory footprint of the solution compared to binarization.

We use the terms *packing* or *bit-packing* to denote the encapsulation of an array of 1-bit values ($+1$'s and -1's) into one 32-bit unsigned integer. For example, if we wish to *pack* a vector $\mathbf{x} \in \{-1, +1\}^{32}$, its *packed* representation, x^p, is given by

$$x^p = \sum_{i=0}^{31} (x_i + 1) 2^{i-1}, \tag{2}$$

where x_i is the ith element of $\mathbf{x}$. This then allows operations such as vector-summations and dot products to be performed using binary (bit manipulation) operations. The dot-product, for example, can be represented as

$$\mathbf{a} \cdot \mathbf{b} = 32 - 2 \times \texttt{popcount}(\texttt{xor}(A, B)), \tag{3}$$

where both A and B are 32-bit unsigned integers holding the packed representations of the vectors $\mathbf{a}$, $\mathbf{b} \in \{-1, +1\}^{32}$. The operation 'popcount' (also known as Hamming weight calculation) is a function for computing the number of bits set to 1, which can essentially simulate vector summation. The operation `xor` in Eq. 3 is the bit-wise 'xor' operation.

C. Acceleration by Bit Manipulation Instructions

Looking at Eq. 3 we see that both 'xor' and 'popcount' are used in inference of binarized CNNs to perform an operation that emulates multiplication for packed weights; this means that both for fully-connected and convolutional layers 'xor'

and 'popcount' are in heavy use and offer a clear optimization target.

The hardware implementation of 'xor' can be found on any programmable processor, whereas a hardware implementation for 'popcount' is mostly available on graphics processing units or CPU SIMD extensions such as ARM NEON. For our target processor, the PULPino microcontroller, the base ISA does not include 'popcount' – this instruction is only present in the *bit manipulation extension* of RISC-V that is still under development [23].

In our experiments, in cases where the target processor did not have a hardware instruction for 'popcount', the LLVM C language description[1] shown in Algorithm 1 was called through __builtin_popcount().

Algorithm 1 LLVM 'popcount', i.e. Hamming weight

```
int32 popcountsi2 (int32 a) {
  uint32 x = (uint32) a;
  x = x − ((x >> 1) & 0x55555555);
  x = ((x >> 2) & 0x33333333) + (x & 0x33333333);
  x = (x + (x >> 4)) & 0x0F0F0F0F;
  x = (x + (x >> 16));
  return (x + (x >> 8)) & 0x0000003F;
}
```

V. EXPERIMENTS

The experimental evaluation of this work consisted of two parts: 1) evaluating the effect of the software-based binarization optimization for our image classifier, and 2) evaluating the effect of the 'popcount' custom instruction on the binarized classifier. Unfortunately, as our ultimate target platform was the PULPino microcontroller for IoT devices, it was not possible to benchmark the original *non-binarized* vehicle classifier on this device as it has no hardware support for floating point computations. Hence it was necessary to use two different target platforms to complete our experiments, and these platforms are summarized in Table II.

The ARM Cortex A53 core is a powerful mobile processor and in our experiments the processor was used under Linux for benchmarking a C language implementation of the original vehicle classifier [20], as well as for the C language implementation of the binarized vehicle classifier.

Experiments on the PULPino microcontroller platform were performed in a simulation environment, which is described next.

A. The ETISS Simulator

The RISC-V ISA is still in a phase of development, as for example the specification is not officially standardized yet. Still, the central components of the specification have matured and have been used to fabricate various chips such as the SiFive FE310 SoC [33]. The application being evaluated in this work however requires the *bit manipulation instruction* extension ('B extension') of the RISC-V ISA. This extension is still in active development [23] and not part of the current specification. Therefore, there is no RISC-V chip available that could be used for evaluating our results, however an alternative way to estimate the performance gain achievable through custom instructions is by simulation.

An RTL (Register-Transfer Level) hardware simulation would not be suitable for fast prototyping as the micro-architecture should be modified to enable the execution of the chosen custom instructions. Additionally, for a time-consuming workload such as our CNN application, the RTL simulation time would be prohibitively high.

The Extensible Translating Instruction Set Simulator (ETISS) focuses on extensibility [34] to support fast prototyping. As ETISS already supports the standard RISC-V base instruction sets, contains a virtual prototype of the PULPino [21] SoC, and allows profiling the application execution time, the use of this simulator was a natural decision our binarized image classifier application.

B. Implementation of the Popcount Instruction

As the PULPino virtual prototype of ETISS currently only supports the RISC-V base ISA, a temporary modification of the virtual prototype was required to enable profiling with support for 'popcount'. From ETISS execution traces it was discovered that the 'xori' instruction of the RISC-V base ISA remained almost unused throughout the whole execution of the binarized vehicle classifier. Therefore, in the PULPino virtual prototype the functional description of 'xori' was modified to provide alternative functionality, i.e. 'popcount', toggled by the value of the 2nd instruction operand.

In the software implementation of the binarized vehicle classifier, the calls to 'popcount' were then replaced with inline assembly calls to 'xori' with the specific operand value that would invoke 'popcount' behavior.

C. Execution Time and Memory Footprint Analysis

Table III shows the experimental results for both A53 and PULPino. From top to bottom the table rows report execution time on A53, execution time on PULPino, data memory footprint, PULPino instruction memory footprint, and CNN classification accuracy.

Looking at the A53 results it can be seen that binarization alone reduced the execution time by more than 80%, and dropped the data memory usage close to 95% when compared to the original floating point C version.

Acceleration by the hardware 'popcount' instruction reduced the computation time of the binarized vehicle classifier by around 55% on the PULPino platform, and also reduced the instruction memory footprint by around 2 kB. The reason for the 55% reduction in execution time can be seen from Table IV that shows the count of executed instructions on the PULPino platform for the binarized vehicle classifier with and without the hardware 'popcount' instruction: the code version that calls the hardware 'popcount' instruction has respectively

[1] https://github.com/sifive/riscv-llvm/blob/master/compiler-rt/lib/builtins/popcountsi2.c

TABLE II
PLATFORMS USED FOR EXPERIMENTS.

Tag	CPU	Platform type	Compiler	Operating System
A53	ARM Cortex A53 (1416 MHz)	Silicon SoC	g++ 5.4.0	Linux Firefly 4.4
PULPino	PULPino (33 MHz)	Virtual prototype on ETISS	riscv32-unknown-elf-gcc 7.1.1	*n/a*

TABLE III
EXECUTION TIME, MEMORY FOOTPRINT AND ACCURACY

Application version Arithmetic	Baseline float32	Binarized int32	Bin+pop int32
A53 Execution time	0.362 s	0.057 s	-
PULPino Exec. time	-	2.62 s	1.18 s
Data Memory	7.2 MB	369 kB	369 kB
Pulpino Instr. Memory	-	21 kB	19 kB
Accuracy [14]	97.09%	92.52%	92.52%

TABLE IV
NUMBER OF EXECUTED INSTRUCTIONS

Instruction name	Binarized int32	Bin+pop int32
lw	8797430	8797417
lbu	272	272
addi	6372539	6354083
slli	2801668	2801668
popcount/xori[2]	4	**3302052**
srli	**16510241**	1
srai	4	4
ori	1	1
andi	3302062	14
sb	268	268
sh	4	4
sw	782165	782109
add	**16704267**	3496013
mul	**0**	0
sub	**3670893**	368845
sll	18632	18632
slt	2553032	2553032
xor	3302048	3302048
or	2451656	2451656
and	**13208192**	0
bne	3232555	3232555
blt	0	0
bge	370058	370058
bltu	4	4
jalr	39	39
jal	57	57
csrrw	1	1
Total	***84078092***	***37830833***

55% less executed instructions. This is because if there is no hardware support for 'popcount', the functionality must be implemented by means of several regular instructions, which can be seen in increased execution counts of 'srli', 'and', 'sub' and 'add' instructions for the binarized version without the hardware 'popcount' instruction. Algorithm 1 shows that these instructions are needed for the software implementation of 'popcount'

The accuracy results shown in Table III are identical to our previous work on binarization that targeted graphics processing units [14].

VI. CONCLUSIONS

In this paper we have presented a convolutional neural network based vehicle image classifier that has been optimized for real-time execution and small memory footprint by a technique called *binarization*. We show that by using 'popcount', a custom instruction in our target processor, the runtime of the binarized image classifier can be reduced by 55%. This result is important due to the fact that 'popcount' has been proposed to be included to a standardized instruction set extension ('B extension') of the recently introduced open source RISC-V instruction set architecture. Besides RISC-V, 'popcount' is already supported in graphics processing units and e.g. in the NEON SIMD extension of ARM processors.

Our work shows that the software-based *binarization* transformation coupled with the hardware-based 'popcount' instruction yields an extremely powerful combination for optimizing inference of convolutional neural networks. Together, the memory footprint is reduced by close to 95%, and execution time is reduced by a magnitude while maintaining an acceptable loss in accuracy. As a results, image classification is performed in 1.18 seconds on the tiny 33 MHz RISC-V microcontroller that is well suited for IoT applications.

As binarization inevitably reduces classification accuracy (most clearly on larger datasets), a potential step for improving accuracy would be the adoption of *heterogeneous bitwidth binarization* [35]. This approach degrades accuracy considerably less than full binarization, already when on average 1.4 bits per weight are used [35].

[2]'popcount' implemented as 'xori' alternative behavior

ACKNOWLEDGMENT

This work was partially funded by the Academy of Finland project 309903 CoEfNet, and by the ITEA3 project 16018 COMPACT (Business Finland diary number 3098/31/2017, German ministry of education and research reference number 01IS17028).

REFERENCES

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

[2] J. Dai, K. He, Y. Li, S. Ren, and J. Sun, "Instance-sensitive fully convolutional networks," in *European Conference on Computer Vision (ECCV)*. Springer, 2016, pp. 534–549.

[3] J. Johnson, A. Karpathy, and L. Fei-Fei, "DenseCap: Fully convolutional localization networks for dense captioning," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 4565–4574.

[4] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *IEEE conference on computer vision and pattern recognition (CVPR)*, 2017, pp. 7263–7271.

[5] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[6] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[7] M. Alioto and M. Shahghasemi, "The Internet of Things on its edge: Trends toward its tipping point," *IEEE Consumer Electronics Magazine*, vol. 7, no. 1, pp. 77–87, 2018.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

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

[10] J. Shen, Y. Huang, Z. Wang, Y. Qiao, M. Wen, and C. Zhang, "Towards a uniform template-based architecture for accelerating 2D and 3D CNNs on FPGA," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2018, pp. 97–106.

[11] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *International Workshop on Frontiers in Handwriting Recognition*, 2006.

[12] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 243–254.

[13] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," in *International Conference on Learning Representations (ICLR) Workshops*, 2018.

[14] M. Khan, H. Huttunen, and J. Boutellier, "Binarized convolutional neural networks for efficient inference on GPUs," in *European Signal Processing Conference (EUSIPCO)*. IEEE, 2018, pp. 682–686.

[15] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," in *British Machine Vision Conference (BMVC)*. BMVA Press, 2014.

[16] 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.

[17] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[18] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[19] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2017, pp. 65–74.

[20] H. Huttunen, F. S. Yancheshmeh, and K. Chen, "Car type recognition with deep neural networks," in *IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2016, pp. 1115–1120.

[21] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Gurkaynak, and L. Benini, "PULPino: A small single-core RISC-V SoC," in *RISC-V Workshop*, 2016.

[22] *The RISC-V Instruction Set Manual*, RISC-V Foundation, 2017, version 2.2.

[23] *RISC-V Bitmanip Extension*, Clifford Wolf, 2019, version 0.37.

[24] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision (ECCV)*. Springer, 2016, pp. 525–542.

[25] F. Pedersoli, G. Tzanetakis, and A. Tagliasacchi, "Espresso: Efficient forward propagation for binary deep neural networks," in *International Conference on Learning Representations (ICLR)*, 2018.

[26] H. Park, D. Kim, J. Ahn, and S. Yoo, "Zero and data reuse-aware fast convolution for deep neural networks on GPU," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 2016, pp. 1–10.

[27] F. Conti, P. D. Schiavone, and L. Benini, "XNOR neural engine: A hardware accelerator IP for 21.6-fJ/op binary neural network inference," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2940–2951, 2018.

[28] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[29] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[30] J.-H. Hoepman and B. Jacobs, "Increased security through open source," *arXiv preprint arXiv:0801.3924*, 2008.

[31] B. Witten, C. Landwehr, and M. Caloyannides, "Does open source improve system security?" *IEEE Software*, vol. 18, no. 5, pp. 57–61, 2001.

[32] C. Cowan, "Software security for open-source systems," *IEEE Security & Privacy*, vol. 99, no. 1, pp. 38–45, 2003.

[33] *SiFive FE310-G000 Manual*, SiFive, Inc., 2017, version v2p3.

[34] D. Mueller-Gritschneder, M. Dittrich, M. Greim, K. Devarajegowda, W. Ecker, and U. Schlichtmann, "The extendable translating instruction set simulator (ETISS) interlinked with an MDA framework for fast RISC prototyping," in *International Symposium on Rapid System Prototyping (RSP)*. IEEE, 2017, pp. 79–84.

[35] J. Fromm, S. Patel, and M. Philipose, "Heterogeneous bitwidth binarization in convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2018, pp. 4006–4015.

PUBLICATION 4

VR-PRUNE: Decidable Variable-Rate Dataflow for Signal Processing Systems

Jani Boutellier; Yujunrong Ma; Jiahao Wu; Mir Khan;

Shuvra S. Bhattacharyya

2022, vol. 70, pp. 1819-1833,

doi: 10.1109/TSP.2022.3162388.

VR-PRUNE: Decidable Variable-Rate Dataflow for Signal Processing Systems

Jani Boutellier, *Senior Member, IEEE*, Yujunrong Ma, Jiahao Wu, Mir Khan, and Shuvra S. Bhattacharyya, *Fellow, IEEE*

***Abstract*—The dataflow concept has been successfully used for modeling and synthesizing signal processing applications since decades, and recently, dataflow has also been discovered to match the computation model of machine learning applications, leading to extremely successful dataflow based application design frameworks. One of the most attractive features of dataflow, especially for signal processing, is related to its formal nature: when properly defined, a dataflow-based application model can be analytically verified for correctness at the stage of application design. This paper proposes VR-PRUNE, a novel dataflow model of computation that is aimed for design of high-performance signal processing software, together with runtime support that allows efficient application deployment to heterogeneous GPU-equipped platforms. Compared to prior work, VR-PRUNE features variable token rate processing, which enables designing adaptive signal processing applications, and implementing solutions that, e.g., allow trading-off between power consumption and filtering bandwidth at runtime. The paper presents the formal concepts of VR-PRUNE, as well as four application examples from domains related to signal processing, accompanied with quantitative results, which show that using VR-PRUNE enables, for example, application power-performance scaling, and on the other hand describing adaptive application behavior with 59% fewer dataflow graph components compared to previous work.**

***Index Terms*—Dataflow computing, design automation, signal processing, parallel processing.**

I. Introduction

DATAFLOW modeling for signal processing systems has been investigated actively since the 1980 s. Many widely used signal processing flavored design frameworks employ dataflow concepts — a couple of prominent examples are GNU Radio [2] and TensorFlow [3]. In the application areas of machine learning (Tensorflow) and software-defined radio (GNU Radio), dataflow features several advantages over conventional, unstructured software design approaches: it provides application modularity, software reuse, concurrency and support for heterogeneous computing.

The dataflow concept however exists in multiple Models of Computation (MoC) that have varying features, especially in terms of *analyzability* and *expressiveness*. For a MoC, analyzability refers to the model's predictability: e.g., a well-analyzable model enables a software compiler to reason about the application's execution flow, apply powerful software optimizations and guarantee absence of deadlocks. Expressiveness, in contrast, refers to the model's flexibility in describing the structure or run-time behavior of an application. In many cases, analyzability and expressiveness are contradictory properties of MoCs.

A key aspect of a dataflow MoC that influences both its expressiveness and analyzability is support for conditional execution — in particular, support for decision making that is required for implementing fundamental *if-then-else* and *for*-loop behavior within the dataflow model. Since many classical signal processing applications behave in a very static fashion (in terms of the rates at which functional modules exchange data), fully static dataflow MoCs, such as *synchronous dataflow* (SDF) [4], have been successfully used also in industrial software (e.g. National Instruments LabView [5] or Keysight SystemVue [6]). However, as algorithms in various signal processing domains, such as wireless communications, video coding and machine learning are exhibiting increasing levels of dynamics and configurability, the need for conditional execution at the dataflow level is becoming important for making modern dataflow frameworks sufficiently expressive. Recent examples of this are *adaptive inference graphs* [7] and *hydra nets* [8] that adaptively switch or skip computations in Convolutional Neural Network (CNN) inference.

In a general sense, such adaptive computation scenarios require the underlying dataflow MoC to support conditional execution, which has traditionally been associated with *dynamic dataflow*, such as *Boolean dataflow* (BDF) [9]. However, it has been shown that the general problem of determining whether a BDF graph can be scheduled for execution with bounded memory, is undecidable [1]. Throughout this paper, the following definition of dataflow graph *consistency* is adopted:

Definition 1 (Consistency): A dataflow graph is consistent if it can be scheduled with guarantees of bounded memory and deadlock-free operation, regardless of what inputs are applied.

Manuscript received June 24, 2021; revised November 12, 2021 and January 26, 2022; accepted March 16, 2022. Date of publication March 25, 2022; date of current version April 26, 2022. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Yuan-Hao Huang. This work was supported by the Academy of Finland under Grant 309903 CoEfNet and the U.S. Army Research Office under Award No. W911NF2110258. *(Corresponding author: Jani Boutellier.)*

Jani Boutellier is with the School of Technology and Innovation, University of Vaasa, 65200 Vaasa, Finland (e-mail: jani.boutellier@uwasa.fi).

Yujunrong Ma and Jiahao Wu are with the Department of Electrical and Computer Engineering, University of Maryland, College Park MD 20742 USA (e-mail: mayu1996@umd.edu; jiahao@terpmail.umd.edu).

Mir Khan is with the Faculty of Information Technology and Communication Sciences, Tampere University, 33014 Tampere, Finland (e-mail: mir.markhan@tuni.fi).

Shuvra S. Bhattacharyya is with the Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park MD 20742 USA (e-mail: ssb@umd.edu).

Digital Object Identifier 10.1109/TSP.2022.3162388

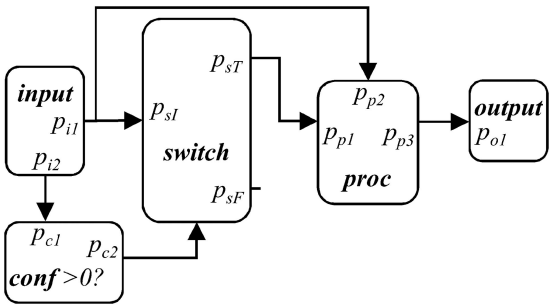


Fig. 1. A weakly consistent dataflow graph. Adapted from [1]. The *conf* actor evaluates sample values coming through port p_{c1}, originating from the *input* actor. *conf* emits Boolean True through port p_{c2} if the sample value is > 0, and False otherwise.

Consider Fig. 1, which shows a dataflow graph with conditional behavior. The *input* vertex produces samples to the three edges that depart from it, whereas the *conf* vertex evaluates whether incoming sample values are greater than zero. In case of a sample value > 0, *conf* emits a Boolean *True* value to its output p_{c2}, otherwise a value of *False*. The *switch* vertex relays the sample coming from p_{sI} to its output p_{sT} if it has received a *True* value from *conf*, otherwise it relays the sample to its output p_{sF}. The graph of Fig. 1 is *weakly consistent* – all samples produced within the graph do not necessarily become consumed, leading to potentially unbounded use of memory: for each sample value ≤ 0 emitted by the *input* vertex, one token is accumulated to the p_{i1} — p_{p2} edge. In other words, unless the *input* vertex doesn't eventually settle into emitting token values > 0, unbounded use of memory will ensue.

It depends on the MoC adopted by the dataflow design framework, how a weakly consistent graph such as the one in Fig. 1 is treated. In a design framework that follows the restricted SDF [4] MoC, the graph could not be modeled at all, as SDF does not allow conditional execution. As another example, in TensorFlow, which is more flexible in this respect, control flow operations [10] would enable implementing the graph, causing version-dependent behavior[1].

The dataflow MoC and design framework VR-PRUNE advocated in this paper addresses the inherent conflict between analyzability and expressiveness in a different way: the VR-PRUNE MoC enables describing conditional execution, however regulated by a set of formal rules and design patterns, which ensure that graph inconsistencies can be detected at design time.

Furthermore, VR-PRUNE features support for variable token rates, which increases the model's expressiveness compared to previous works. Consequently, in this paper we show that VR-PRUNE offers a theoretically solid dataflow basis for future programming frameworks similar to what TensorFlow and GNU Radio are at the moment.

Some of the main ideas of VR-PRUNE were briefly presented in [11] recently. This full-length article extends the conference paper by

- A complete theoretical presentation of the VR-PRUNE Model of Computation,
- Design rules and patterns that ensure compile-time consistency analysis of VR-PRUNE application graphs,
- An open source[2] runtime framework *VPRF* that has been constructed around VR-PRUNE concepts, and
- Run-time experiments on heterogeneous desktop and embedded platforms, which highlight the efficiency and expressiveness of VR-PRUNE.

The rest of this paper is organized as follows: Section II presents related works, Section III describes the proposed VR-PRUNE Model of Computation, Section IV presents the formal VR-PRUNE design rules, Section V discusses consistency of VR-PRUNE graphs, Section VI shows the experimental results related to VR-PRUNE, Section VII discusses the proposed work, and Section VIII concludes the paper.

II. Background

In the dataflow programming concept, applications are expressed as directed *graphs*. The application graph consists of nodes, which are called *actors* that are interconnected by *edges*. Edges carry data that is encapsulated within *tokens*; for example, in an image processing application, a single token may encapsulate the data related to one image pixel, or the data of a complete frame, depending on the application.

The interconnection between an actor and an edge is called a *port*. A port that is connected to an edge, which departs from an actor is called an *output port*, and respectively a port that is connected to an incoming edge is called an *input port* of an actor. Each port is associated with a non-negative integer value called *token rate* that determines how many tokens the actor consumes (for input ports) from its associated edge, or how many tokens the actor produces (for output ports) to its associated edge upon one *firing*.

Firing is the dataflow concept for computation. Consider a simple actor a that has the mere purpose of dividing one integer value with another integer. Logically, a should have two input ports, one for the divisor and one for the dividend, as well as one output port for the result. In order to perform the computations related to a division operation, a needs to have one token available from the port that provides the dividend value, and one token from the port that provides the divisor value. Hence, we can say that for the dividend port a has a fixed input token rate of one, which is also the case for the divisor port and the result port.

Another well-known model for describing information flow in dynamic discrete systems is Petri nets [12]. The main advantage of dataflow graphs over Petri nets is their compactness and convenience in mapping real-life applications to graph-based models [13]. However, a significant class of dataflow models can be transformed to Petri nets, thus enabling application of analytic methods that have been designed for Petri nets, to dataflow models [14].

Most of the differences between existing dataflow MoCs associate with rules and restrictions related to token rates. *Homogeneous* dataflow MoCs require that every port of each actor has

[1] In TensorFlow 1.14 the execution of the graph caused a runtime error, whereas in 2.3.1 the *sink* vertex received 0 samples.

[2] Available at https://gitlab.com/jboutell/vprf

strictly a token rate of one, whereas the synchronous dataflow (SDF) [4] MoC allows a fixed positive integer token rate for each port. Examples of design frameworks that are based on SDF are PREESM [15] and StreamIt [16], [17]. Cyclo-static dataflow (CSDF) [18], on the other hand, enables token rates to change in fixed, periodic cycles. SDF (and its homogeneous variant) and CSDF are regarded as *static* dataflow MoCs, as their token rates are completely predetermined at application design time. Consequently, the dataflow graphs are analyzable at compile-time, enabling formal proofs for absence of deadlocks and bounded memory.

Somewhat more flexibility can be added to the SDF MoC by adding *scenarios*, different SDF graph topologies that are at runtime switched, e.g., based on a finite state machine. The baseline work in this direction is SADF [19], and recently also full software design frameworks such as HOPES+ [20] have been built around the SADF concept.

Dynamic dataflow MoCs, of which *dataflow process networks* (DPN) [21] is one of the most well-known examples, allow port token rates to change arbitrarily at application run time, and therefore possibilities for graph consistency verification at design time are very limited. In the past decade, dynamic dataflow around the CAL language [22] and its subvariant RVC-CAL [23] has triggered the development of several design frameworks such as Tÿcho [24], Orcc [25] and SHeD [26].

Between the extremes of fully static and fully dynamic dataflow exist a number of MoCs that balance between analyzability and expressiveness. *Well-behaved dataflow* [27] restricts dynamic application behavior to take place within subgraphs that follow a predefined topology. These subgraph templates enable expressing conditional constructs such as if-then-else and loops at the dataflow level, while still guaranteeing finite-time verification for bounded memory. Our recent work PRUNE [28] elaborated the ideas of WBDF into a MoC accompanied with a high-performance runtime framework for heterogeneous computing, and design time algorithms for verifying graph consistency.

Another branch of work in boundedly dynamic dataflow is *variable-rate dataflow* (VRDF) [29]. The paper [29] presents a dataflow MoC that allows non-negative integer port token rates that can vary between arbitrary pre-defined limits (called *variable-rate* from here on), and an algorithm that computes the memory capacity required to execute the graph. Additionally, the paper [29] describes a check procedure for determining if a given graph is valid for memory capacity computation. The proposed VR-PRUNE MoC adopts the concept of variable token rates from VRDF, however the emphasis of VR-PRUNE is on high-performance processing on heterogeneous platforms, and consequently VR-PRUNE introduces the restriction of *symmetric token rates*, preventing direct adoption of VRDF models to VR-PRUNE.

Table I summarizes related dataflow MoCs and frameworks detailing their features: *decidable* indicates whether graph consistency analysis is a decidable problem, *dynamic* expresses whether the model supports port token rates that can at run time vary according to a two-valued function, *high-performance* tells whether a design framework with performance metrics has been published, and *variable-rate* shows whether the model permits variable-rate port token rates. Variable-rate dataflow can be considered as a generalization of two-valued dynamic dataflow, and in Section VI we will show that variable-rate dataflow provides a higher degree of expressiveness in terms of describing the same application behavior with considerably fewer dataflow graph elements than two-valued dynamic dataflow.

TABLE I
COMPARISON TO RELATED DATAFLOW MODELS AND LANGUAGES

Work	Decidable	Dynamic	High-performance	Variable-rate
SADF [19]	+	+	-	+(-)
BDF [9]	-	+	-	-
DAL [30]	-	+(-)	+	+(-)
RVC-CAL [25]	-	+	+	+
StreamIt [17]	+	-	+	-
PRUNE [28]	+	+	+	-
VRDF [29]	+	+	-	+
VR-PRUNE	+	+	+	+

Cases requiring explanation in Table I are: SADF [19] – variable-rate dataflow is in principle possible, but this requires one dataflow graph per specific token rate making the solution impractical for larger rate variations; DAL – dynamic and variable-rate dataflow can be built on top of the framework, but not for GPU-accelerated graph components.

The proposed VR-PRUNE Model of Computation builds on concepts introduced in VRDF [29], WBDF [27], and our previous work PRUNE [28]. Consequently, the VR-PRUNE MoC features

- A dataflow model that supports dynamic token rates,
- Token rate variability within pre-defined limits, and
- Design time analysis for bounded memory and absence of deadlocks through a set of design rules and patterns.

In Section VI of this paper we show that token rate variability enables 1) expressing data dependent graphs in a more compact representation, 2) capturing application behavior that was not possible in our previous work PRUNE. Additionally, we show that the increased flexibility of VR-PRUNE does not add computational overhead compared to PRUNE. The following section formally presents the VR-PRUNE MoC.

III. PROPOSED MODEL OF COMPUTATION

In this section, the components of VR-PRUNE graphs are introduced together with an example graph.

A. Notation and Port Types

Following the notation of our previous work [28], a VR-PRUNE application graph $G = (A, F)$ consists of a set of actors A, and a set of directed edges F that interconnect the actors of A. By definition of dataflow, the edges follow first-in-first-out (FIFO) communication behavior, and for this reason we interchangeably also refer to edges as FIFOs. Actors connect to edges over ports, which are classified into *input ports* (for ports that consume tokens) and *output ports* (for ports that produce tokens). Each actor $a \in A$ can contain any non-negative number of input and output ports, and $a = parent(p)$ denotes an actor a contains port p. More briefly, this can also be expressed by p^+_{a1}, which refers to the first output port of actor a (Note: the subscript

does not indicate any form of multiplication, only indexing). The superscript $+/-$ corresponds to output/input port respectively.

VR-PRUNE ports consist of three different port types: each port is either a (input or output) *control port*, *static regular port (SRP)* or *dynamic regular port (DRP)*. A SRP p has a fixed token consumption/production rate $atr(p)$, which is set at application design time. A DRP p, however, has a variable token rate that is defined as $lrl(p) \leq atr(p) \leq url(p)$, where non-negative integers lrl, atr and url stand for lower rate limit, active token rate, and upper rate limit of p, respectively. lrl and url are values that are fixed at design time, whereas atr may vary within the limits of lrl and url at run time. Finally, a control port p must have a fixed token rate of 1.

Each FIFO $f \in F$ is connected to exactly one output port p^+ of actor $parent(p^+)$ and to exactly one input port p^- of $parent(p^-)$. Moreover, $fifo(p_a^+)$ and $fifo(p_b^-)$ refer to the FIFO connected to ports p_a^+ and p_b^-, respectively. Ports p_a^+ and p_b^- are *connected* when $fifo(p_a^+) = fifo(p_b^-) = f$, where actors a and b are referred as the source ($source(f)$) and sink ($sink(f)$) of the same FIFO f. In VR-PRUNE, connected ports must always be of the same port type, and a valid VR-PRUNE graph is not allowed to have unconnected ports, as for example the port F of Fig. 1.

In the VR-PRUNE MoC, an output port p^+ that is a control port or SRP, can be connected to multiple FIFOs, but in this case each FIFO must have a unique source and sink port, and every input port p^- must have only one FIFO connected to it. If an output port p^+ is connected to multiple input ports $p_i^-, i = 1 \ldots K$ in the aforementioned way, then each p_i^- must be of the same port type as p^+.

The current number of tokens in FIFO f is denoted as $tokencount(f)$, the FIFO's token capacity is given by $capacity(f)$, and $delay(f)$ denotes the number of delays (initial tokens) in f.

Similar to its predecessor PRUNE, VR-PRUNE adopts the concept of *symmetric-rate dataflow*, which requires for connected ports p_a^+ and p_b^- that $atr(p_a^+) = atr(p_b^-)$. A VR-PRUNE actor a can fire when a) for each input port p_a^- holds $tokencount(fifo(p_a^-)) \geq atr(p_a^-)$, and b) for each output port p_a^+ holds $capacity(fifo(p_a^+)) - tokencount(fifo(p_a^+)) \geq atr(p_a^+)$.

B. Actor Types

Each actor in a VR-PRUNE graph G must fall into one of the four actor types that are characterized by numbers, types and directions of ports, as described below. If an actor does not meet the requirements of any of the four actor types, the actor cannot be included into VR-PRUNE graph G.

1) Static Processing Actor (SPA): The ports of SPA actors can only be of the type SRP, and therefore an SPA actor can be understood to operate similar to an actor of the SDF [31] MoC.

2) Dynamic Actor (DA): A DA has at least one DRP, at least one input control port, and any non-negative number of SRPs. All the DRPs of DA x need to be either of input direction, or of output direction as required by the design rules (Section IV). This restriction enforces modularity of VR-PRUNE graphs and acts as a necessary condition for analyzability. Furthermore, the number of DRPs in x must be greater or equal to the number of its input control ports, since each DRP of x is controlled by exactly one input control port of x.

When a DA x first fires, x first consumes one token from each of its control ports. The values of these consumed tokens set the $atr(p_x)$ for each DRP p_x of x. After the *atr*s of each DRP p_x have been set, firing of x proceeds by following usual dataflow semantics: tokens are consumed from the input ports of x according to the port-specific *atr* values, and consequently tokens are produced to the output ports of x following the port-specific *atr* values.

Fig. 2 shows a VR-PRUNE subgraph with one configuration actor q and two dynamic actors, x and y. The figure illustrates how the token values originating from black-colored input control ports p_{x1} and p_{x2} associate with the DRPs p_{x3}, p_{x4} and p_{x5} in a one-to-many relationship. For actor x that has a DRP p, $cport(p)$ is the input control port of x that is associated with p. Drawing an example from Fig. 2, $cport(p_{x3}) = p_{x2}$.

3) Dynamic Processing Actor (DPA): DPAs are required to have at least one input DRP, at least one output DRP, at least one control input port, and any number of SRPs. In the beginning of firing DPA a, a first consumes one token from each of its control ports. The values of the tokens originating from the control ports set the atr for each DRP of a. Similar to DAs, the number of DRPs must be greater or equal to the number of input control ports, and firing of a proceeds by consuming $atr(p_{ai}^-), i = 1, 2, \ldots, K$ tokens from each of a's K input (SRP or DRP) ports, finally producing $atr(p_{aj}^+), j = 1, 2, \ldots, L$ tokens to each of a's L output (SRP or DRP) ports. Evidently, FIFO f that is connected to DRP p must have a token capacity of at least $url(p)$: $capacity(fifo(p)) \geq url(p)$.

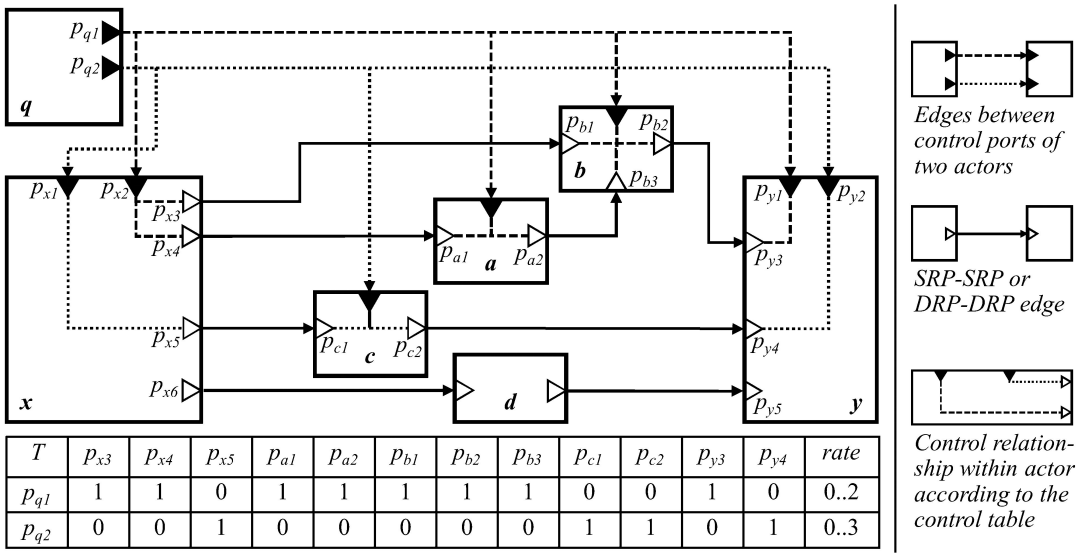
In Fig. 2, a, b and c are DPAs. Out of these, b has two input DRPs (p_{b1} and p_{b3}) and one output DRP (p_{b2}), all of which are controlled by the single input control port of b.

4) Configuration Actors: A configuration actor (example: q in Fig. 2) must have one or more output control ports, which are required, by definition, to have a token rate of unity. Additionally, a configuration actor can have zero or more data ports, which are SRPs. The data ports can either have input or output direction.

The tokens produced by the output control ports contain non-negative integer values that define port-specific atrs for DAs and/or DPAs that consume the control tokens. The relationships between output control ports of configuration actors and input control ports of DAs/DPAs are unambiguously defined by a *control table* that is described below.

C. Control Table and Firing

In the VR-PRUNE MoC, variable token rates are restricted to subgraphs called Dynamic Processing Graphs (DPGs) that define graph-level structure for ensuring analyzability. The VR-PRUNE concept allows various DPG types, however in this paper we define one specific DPG type in Section V, which is suitable for all application examples presented in Section VI. DPG types may impose additional restrictions to actor types, actor port counts, or connectivity between actors.



T	p_{x3}	p_{x4}	p_{x5}	p_{a1}	p_{a2}	p_{b1}	p_{b2}	p_{b3}	p_{c1}	p_{c2}	p_{y3}	p_{y4}	rate
p_{q1}	1	1	0	1	1	1	1	1	0	0	1	0	0..2
p_{q2}	0	0	1	0	0	0	0	0	1	1	0	1	0..3

Fig. 2. VR-PRUNE subgraph example. An example of control relationships between a configuration actor (q), dynamic actors (x, y), and DPAs (a, b, c) expressed equivalently by a control table, and graphically. Note: the *rate* column of the control table refers to the range of valid atr values (defined by lrl and url) emitted by the output control port.

The complete VR-PRUNE application graph G can consist of any number of interconnected DPGs: dynamic actors and configuration actors of a DPG are allowed to connect outside the DPG over SRPs without restrictions. In fact, using the concept of hierarchical dataflow graphs (e.g. [32]), a DPG could be represented as a composite actor with static token rate ports. Formal presentation of hierarchical graphs is however limited outside the scope of this work as clustering of actors may change the graph semantics and cause deadlock [32]. In the rest of this paper the analysis concentrates on internal behavior of individual DPGs.

Fig. 2 shows an example of a DPG, where q is a configuration actor, a, b and c are DPAs, x and y are DAs, and d is an SPA. Each DPG is associated with a *control table* T that unambiguously defines 1) the control relationships between output control ports of configuration actors, and DRPs of DAs and DPAs, and 2) sets limits for variable token rates by means of output control port specific lrls and urls.

The control table of Fig. 2 is shown below the DPG — it is a matrix with dimensions $h \times (w + 1)$, where h and w equal the number of output control ports and DRPs, respectively. In the control table, a value of '1' indicates a control relationship between the corresponding output control port (row) and DRP (column), whereas a value of '0' indicates that the control port and DRP are not related. Since each DRP is required to be controlled by exactly one output control port of a configuration actor, a valid control table must have a column sum of '1' for each DRP column. The lrl and url values are defined per output control port in the last column of the control table, and apply to all DRPs that are associated ('1') with that row.

Finally, we define the meaning of a *complete cycle* related to a DPG in the spirit of [9]: assuming that a DPG is consistent (as discussed Section V), we define a complete cycle of a DPG as a sequence of actor executions that returns the DPG to its original state. The execution of a complete cycle S of actors of the form $S = q_1, q_2, \ldots, q_m, a_1, a_2, \ldots, a_n$, where the q_i's are the control actors of the DPG, and the a_i's are the SPAs together with the active DAs and DPAs of the DPG, as determined by the execution sequence $q_1, q_2, \ldots, q_m$. Here, an *active* DA or DPA a means that (1) a has input tokens on all control ports, and (2) a is configured by the incoming control tokens so that at least one DRP of a will have a nonzero rate on the next actor firing. To be valid, a complete cycle must also satisfy the condition that there is no FIFO buffer underflow within the edges of the DPG when executing S.

IV. Design Rules

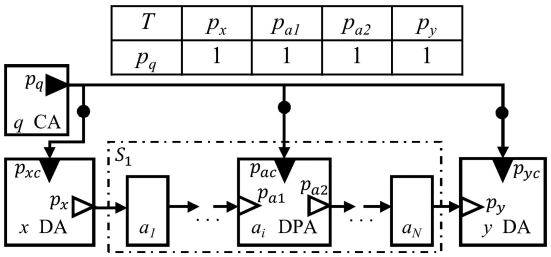
This section presents the VR-PRUNE design rules, which apply to all types of DPGs, providing generic restrictions for supporting dataflow consistency. The approach of using design rules is similar to previous works [27], [28], [32]. Specific types of DPGs may impose further design restrictions beyond VR-PRUNE design rules. Before presenting the rules, some mandatory definitions are provided.

We define two actors, a and b, as *adjacent*, if at least one port of a is connected with at least one port of b.

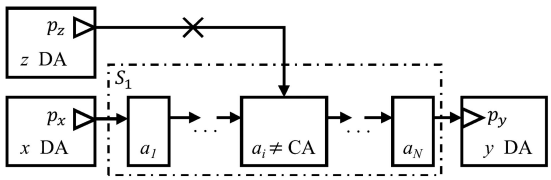
Definition 2 (Chain): A *chain* is a non-empty sequence $S = (a_1, a_2, \ldots, a_N)$ of actors, such that $\forall i = 1, 2, \ldots, N$, a_i and a_{i+1} are adjacent.

Consequently, chain S connects a_1 and a_N. Furthermore, if all a_i are distinct, then we call S a *simple chain*.

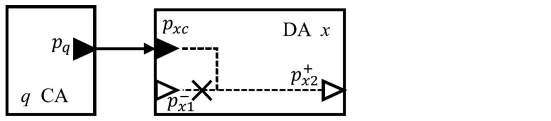
Suppose p_x and p_y are distinct ports of two actors x and y, respectively. We say that p_x and p_y are *linked ports* if (a) $fifo(p_x) = fifo(p_y)$, or (b) there is a simple chain $(x, a_1, \ldots, a_N, y)$ such



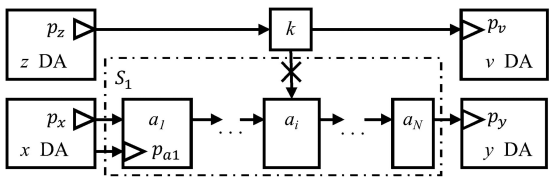
(a-b) Linked port control rule and Balanced delay rule.



(c) Connecting subchain rule.



(d) Single-sided dynamism rule.



(e) Encapsulation rule.

Fig. 3. VR-PRUNE design rules. Illustrations related to design rules (a)–(e). Solid black triangles denote control ports.

that p_x connects with a port of a_1 and p_y connects with a port of a_N. Note that for the linked ports $\{p_x, p_y\}$, there can be multiple *connecting subchains* $a_1, \ldots, a_N$. If p_x and p_y are linked ports, and they are both DRPs, then we say that they are *linked DRPs*.

For all design rules (a)–(e): Suppose $\{p_x, p_y\}$ are linked DRPs whose parents are dynamic actors x and y, and $S_1 = (a_1, a_2, \ldots, a_N)$ is a connecting subchain associated with $\{p_x, p_y\}$:

a) Linked port control rule: Each pair of linked DRPs $\{p_x, p_y\}$, and each DRP within actors of S_1 must be controlled by the same output control port p_q.

b) Balanced delay rule: The input control ports associated with the linked DRPs $\{p_x, p_y\}$, and each DRP within actors of S_1 must be connected to p_q with the same delay. In other words – suppose p_a is a DRP of $a \in S_1$, then $delay(p_q, cport(p_x)) = delay(p_q, cport(p_y)) = delay(p_q, cport(p_a))$.

c) Connecting subchain rule: The actors $a_i, i = 1, 2, \ldots, N$ must all be of type SPA or DPA, and $a_i \in S_1$ may not belong to any connecting subchain $S_2 = (b_1, b_2, \ldots, b_M)$ that is associated with a dynamic actor $z \notin \{x, y\}$.

d) Single-sided dynamism rule: The DRPs of actor x are only allowed to have output direction, and the DRPs of actor y are only allowed to have input direction.

e) Encapsulation rule: Suppose $k \notin S_1$ is an actor that connects to $a_i, i = 1, 2, \ldots, N$. 1) If $k \in \{x, y\}$, then a_i may connect to k only via a DRP p_{a1}. 2) If $k \notin \{x, y\}$, then k must belong to another connecting subchain $S_2 = (b_1, b_2, \ldots, b_M)$ associated with a pair of linked DRPs $\{p_{x2}, p_{y2}\}$ such that $parent(p_{x2}) = x$ and $parent(p_{y2}) = y$.

Fig. 3 illustrates the design rules (a)-(e) as follows: The topmost subfigure is a joint example of rules (a) and (b): the dynamic actors x and y are connected by the chain $S_1 = a_1, \ldots, a_i, \ldots, a_N$, and consequently p_x, p_y and each DRP of each DPA within S_1 need to be controlled by the same control output port p_q of control actor q. Moreover, each FIFO $\{p_q, p_{xc}\}, \{p_q, p_{yc}\}$, as well as $\{p_q, cport(p_{a_i})\}$, where p_{a_i} is a DRP of actor $a_i \in S_1$, need to have the same number of delay tokens.

Subfigure (c) of Fig. 3 illustrates the Connecting subchain rule: on one hand, actor a_i of S_1 is not allowed to be a control actor (CA), and on the other hand a_i is not allowed to be part of any connecting subchain that is associated with dynamic actors other that x and y, such as z in the figure.

Violation of the Single-sided dynamism rule (Subfigure d) is illustrated by a dynamic actor x, which incorrectly features both an input DRP p_{x1}^-, and an output DRP p_{x2}^+.

Finally Subfigure (e) of Fig. 3 depicts an actor k that does not belong to the connecting subchain between linked DRPs $\{p_x, p_y\}$. The actor k is part of a connecting subchain that is associated with a different pair of dynamic actors, z and v, which is disallowed. Additionally, Subfigure (e) also illustrates (observe actor a_1) the case where k is one of the two dynamic actors that are interconnected by S_1. In this case, an actor $a_1 \in S_1$ is not allowed to connect to the dynamic actor via a port p_{a1} of type SRP — in contrast, the connection is allowed if p_{a1} is of type DRP.

V. Consistency

This section first defines the Dynamic Processing Graph type used throughout the rest of this paper, and consequently shows that determining its consistency is a decidable problem.

A. The Switch Type Dynamic Processing Graph

The previously introduced VR-PRUNE MoC (Section III) and design rules (Section IV) have been defined without unnecessarily strict constraints, not to limit expressiveness or future developments that build on the MoC. However, for ensuring decidable consistency analysis, additional constraints might need to be incorporated to specific DPG types. Next, we introduce the *Switch DPG* (sDPG) type that is a restricted type of DPG, however generic enough for capturing all the application use cases presented in Section VI. In addition to enabling analysis of the application use cases in our experimental study, the developments we present involving the sDPG demonstrate how groups of relevant applications can be represented and formally analyzed by formulating suitable constraints within the VR-PRUNE modeling framework.

Each sDPG consists of a) exactly one configuration actor q, b) exactly two dynamic actors, x and y, and c) any positive number of *linked DRPs* that connect x and y. These restrictions a)-c) are only associated with the sDPG type, and together with the VR-PRUNE design rules ensure decidable consistency analysis for sDPGs. Other types of DPGs with different restrictions would consequently require a separate consistency analysis procedure.

The linked DRPs of an sDPG establish zero or more connecting subchains between x and y. These chains form the *dynamic components* (DCs) of the sDPG. Given an sDPG D, the set of DCs of D is denoted as $Z_c(D)$, and the pair of dynamic actors in D is denoted $\delta(D) = \{x, y\}$. Fig. 2 shows an example of such an sDPG with actors q, x and y. The DCs of this graph are explained in the end of this subsection.

Consider an sDPG D that contains dynamic actor x with K output DRPs p_{xi} $(i = 1, 2, \ldots, K)$, and dynamic actor y with L input DRPs p_{yj} $(j = 1, 2, \ldots, L)$. We require that each p_{xi} is a linked DRP with at least one of p_{yj}. Our procedure for finding the DCs $Z_c(D)$ associated with a given DPG D can be expressed as follows:

1) For each SRP p_a of actor $a \in \{q, x, y\}$, remove *fifo*(p_a). Next, remove all actors and edges that have become disconnected from the set of actors $\{q, x, y\}$ through the preceding removal of FIFOs *fifo*(p_a).

2) For each linked DRP $\{p_{xi}, p_{yj}\}$, where *fifo*$(p_{xi}) =$ *fifo*(p_{yi}), insert a dummy actor d (DPA) such that *fifo*$(p_{xi}) =$ *fifo*(p_d^-) and *fifo*$(p_d^+) =$ *fifo*(p_{yj}).

3) Remove q, x, y, and all FIFOs *fifo*(p_q), *fifo*(p_x) and *fifo*(p_y) in D. This removal procedure decomposes D into a set of connected components that form the DCs. Thus, $Z_c(D) = \{Z_1, Z_2, \ldots, Z_M\}$, where $M \in [1, min(K, L)]$ is an integer constant.

For an sDPG D to be *valid*, 1) each DPA a within each $Z_k, k \in [1, M]$ of D must have exactly one control port p_a, and 2) within D, no *fifo*(p), where p is a DRP, may have $delay($*fifo*$(p)) > 0$.

Fig. 2 depicts an example of a valid sDPG D. Following the above described three-stage procedure for discovering DCs, the resulting DCs are $Z_c(D) = \{Z_1, Z_2\}$. The actors related to the DCs are all DPAs, such that $Z_1 = \{a, b\}$, $Z_2 = \{c\}$. Notice that the SPA actor d was removed in the 1st stage, and is not part of the DCs. As required by the design rules (Section IV) and sDPG validity requirements, both DPAs a and b of Z_1 have one input control port each, which is connected to p_{q1}. The control port of actor c (which belongs to DC Z_2) is connected to p_{q2}, and consequently the token rates of ports related to actors within Z_1 and Z_2 can vary independently of each other.

B. VR-PRUNE Graph Consistency

VR-PRUNE graphs may consists of several DPGs, but our design rules ensure that the individual DPGs are independent of each other. Since the existence of DRPs (and hence non-static token rates) is limited to within DPGs, a) the actors outside DPGs and b) ports of DPG actors connecting outside the DPGs necessarily have static token rates. Hence, the consistency of the VR-PRUNE application graph G can be validated using standard SDF validation techniques [31]. Therefore, we limit our discussion on consistency to within sDPGs.

In the following, a proof for the decidability of the consistency analysis of sDPGs is presented. The reasoning followed by the proof is to show that 1) the configuration actor of the sDPG fixes the token rate of each DRP within the sDPG for the duration of one complete cycle of an sDPG, 2) consequently, each DC of the sDPG can be interpreted as a fixed token rate (SDF) graph for the duration of that complete cycle, and 3) finally, the whole sDPG can be considered as an SDF graph, for which it is well-known [4] that determining consistency is a decidable problem.

Lemma 1: Assuming all DAs and DPAs are contained within sDPGs: if all sDPGs of a VR-PRUNE graph G are consistent, then the whole VR-PRUNE graph G is consistent.

Proof: Let $Z_c(D) = Z_1, Z_2, \ldots, Z_M$ be the set of DCs of a valid sDPG D. The actors of DCs are by the Connecting subchain rule required to be of type DPA or SPA.

Since we only consider valid sDPGs, each DRP of $actors(Z_k)$ within a single DC $Z_k, k \in [1, M]$ is controlled by the same output control port p_q of configuration actor q. Consequently, p_q sets the atr of all DRPs within $actors(Z_k)$ to the same value for each complete cycle of an sDPG, and Z_k can be considered as an SDF graph. Since there is a finite maximum of $url - lrl + 1$ different token rates per DC, and, considering that the set of DCs within one sDPG is finite, it is decidable to determine whether or not the sDPG is consistent.

If all the Z_k's are consistent, then there exists a valid, periodic schedule $P(Z_k)$ for each Z_k [4]. $P(Z_k)$ defines a schedule for each actor, $actors(Z_k)$, related to the current atr set by p_q that is associated with Z_k.

Considering $actors(Z_k)$, for each FIFO f connected an actor $a \in actors(Z_k)$, there exists a buffer bound $B_k(f)$ that indicates the maximum token count on f at any stage of $P(Z_k)$ execution. This buffer bound exists as a consequence of SDF graph consistency [4]. Among all DCs Z_c, there is a finite FIFO-specific bound $\beta(f) = max(B_k(f) \mid Z_k \in Z_c(D))$.

The whole sDPG can then be executed by a sequence of schedules $\Omega = (O_1, O_2, \ldots)$ such that for every O_k, there is an $H_k \in Z_c(D)$, where $O_k = P(H_k)$. H_k is the kth executed DC within the sDPG.

As each Z_k is assumed to have a valid, periodic schedule, execution of O_k does not cause a net change to the token counts of the FIFOs between $actors(Z_k)$. Therefore, the token count of FIFO f is bounded by $B_k(f)$. Finally, the token count of f is during execution of Ω is limited to $\beta(f)$. □

VI. Experimental Results

Experiments related to VR-PRUNE were performed on four application use cases: 1) adaptive digital predistortion, 2) parallel image classification, 3) dynamic-update digital predistortion and 4) object detection. Examples 1 and 3 concern real-time signal processing for wireless communications, whereas examples 2 and 4 concentrate on deep convolutional neural networks.

The experimental results show that the increased expressiveness of VR-PRUNE (compared to PRUNE [28])

- Enables expressing the same application structure by clearly fewer dataflow graph elements,
- Enables describing dataflow behavior that cannot be captured by PRUNE,

TABLE II
PLATFORMS USED FOR EXPERIMENTS

Tag	CPU	GPU	Operating System
i7	Intel Core i7-8650U: 1.9 GHz, 4(8) cores	Intel UHD Graphics 620	Ubuntu Linux 18.04
N2	Amlogic S922X (1.8 GHz, 4×ARM Cortex-A73 + 2×ARM Cortex-A53 cores)	ARM Mali G-52	Ubuntu Linux 18.04
XU3	Samsung Exynos 5422 (2.1 GHz, 4×ARM Cortex-A15 + 4×ARM Cortex-A8 cores)	ARM Mali T628	Ubuntu Linux 14.04

- Provides means for saving power by adaptive reduction of computational effort, and
- Does not cause excessive computational run-time overhead.

The first point listed above results because VR-PRUNE enables more compact representation of design functionality. For elaboration on the importance of using compact representations in system-level modeling, see for example [33]. The run-time measurements have been performed on three platforms (see Table II), one of which is a regular mobile workstation (i7), and the other ones are embedded platforms (XU3 and N2), all equipped with GPUs.

A. Run-Time Framework and Application Programming

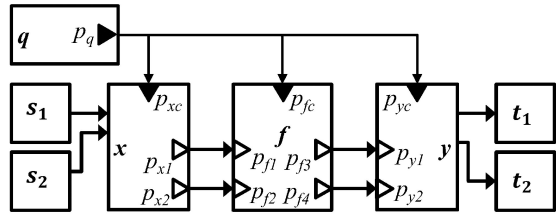
In order to conduct run-time experiments for measuring the efficiency of VR-PRUNE, the PRUNE runtime framework was extended to accommodate variable token rates. We refer to the resulting extension as the VR-PRUNE Runtime Framework (VPRF).

VPRF operates under Linux, and bases its concurrent computation infrastructure on Linux parallel computing primitives: each actor is instantiated as a separate thread that can either be assigned to a specific processor core, or be auto-assigned to a free core by the operating system. FIFOs are implemented as memory arrays, and read/write access to them is arbitrated by *mutex* constructs. Although Section V discussed execution schedules of VR-PRUNE graphs for the purpose of consistency analysis, VPRF follows static assignment scheduling [34] where the operating system determines the execution order of actors at run-time, subject to dataflow constraints.

VPRF also features deeply embedded support for interfacing GPUs. This means that the FIFO and actor primitives of VPRF have been designed from the beginning for efficient data transfers between CPU cores and the GPU, as well as data exchange between GPU kernels, including functionality for variable-length data transfer to/from GPU. VPRF also includes a prototype graph validity checker that inspects application graphs against VR-PRUNE design rules. The dataflow models and experiments described in Sections VI-B and VI-C have been done under VPRF.

The VPRF application programmer writes the application code using the C language or OpenCL for actor descriptions, and XML for describing the application graph. VPRF follows a predefined actor template that originates from the DAL [35] framework: each actor has *initialize*, *fire* and *finish* functions, as well as a persistent actor-specific data structure for preserving actor state between firings. For actors that are aimed to GPU execution, the actor behavior is expressed in the OpenCL language (OpenCL was selected over CUDA for reasons of wider hardware support, as almost all CUDA devices support OpenCL, but not vice-versa). VPRF provides a compact set of

(a) VR-PRUNE graph of adaptive digital predistortion.



T	p_{x1}	p_{x2}	p_{f1}	p_{f2}	p_{f3}	p_{f4}	p_{y1}	p_{y2}	*rate*
p_q	1	1	1	1	1	1	1	1	1..5

(b) PRUNE graph of adaptive digital predistortion.

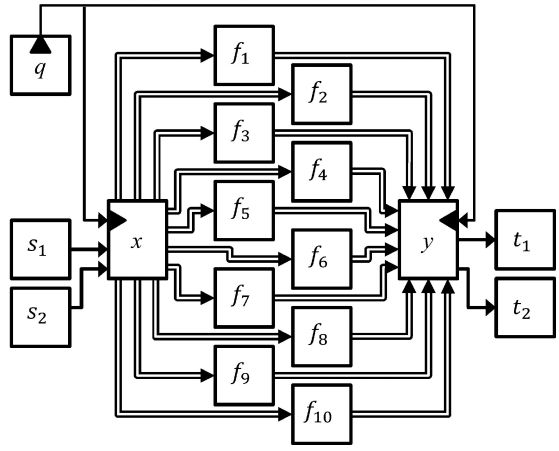


Fig. 4. The adaptive digital predistortion application. Subfigures: (a) VR-PRUNE graph, (b) PRUNE graph. In Subfig. (b) double-line arrows depict 2×FIFOs.

function calls for inter-actor data exchange over FIFO buffers. The function calls effectively hide the complexity related to GPU programming from the programmer: data transfer from a CPU-based actor to a GPU-based actor is carried out using the same function as data transfer between two CPU-based actors. VPRF includes a compiler that transforms the application graph XML file into a top-level C file, which can together with the actor descriptions be compiled into an executable.

Exploiting target platform heterogeneity and parallelism is an important factor for efficient signal processing. Here, VPRF follows the approach first introduced by DAL [35], and later used in PRUNE: using a *mapping specification* (XML file) the application programmer can assign each actor to a specific CPU core, or to the system's GPU. If an actor is assigned to the GPU,

the programmer also needs to indicate the number of *work items* and *work groups* as required by OpenCL [36]. If the work size of an actor exceeds the maximum work size of the GPU, the underlying OpenCL driver automatically divides the processing into multiple passes. On the other hand, if the actor work size is less than the GPU's maximum work size, the GPU runs with reduced utilization; simultaneous execution of several actors on the same GPU is not supported.

CPU core assignments are handled by Linux CPU affinity masks that define which CPUs/cores are eligible to execute each thread (actor). Mapping an actor to the GPU requires expressing the actor's internal behavior in OpenCL, but data exchange between the GPU and CPU, as well as GPU initialization are handled by VPRF.

TABLE III
ADAPTIVE DIGITAL PREDISTORTION LINES OF CODE PER ACTOR

Framework	s	q	x	f	y	t	mean
PRUNE	59	64	139	44	141	48	**83**
VR-PRUNE	59	68	103	64	106	48	**75**

TABLE IV
ADAPTIVE DIGITAL PREDISTORTION THROUGHPUT IN COMPLEX FLOAT MEGASAMPLES/S, AS A FUNCTION OF ENABLED FILTER BRANCHES (%), ON THE I7 CPU AND ON THE N2 CPU. HIGHER IS BETTER

Framework	20%	40%	60%	80%	100%
PRUNE i7	**90.77**	45.75	30.50	22.60	18.29
VPRF i7 (prop.)	88.44	**46.86**	**31.62**	**23.63**	**18.68**
PRUNE N2	17.88	9.53	6.50	4.88	4.11
VPRF N2 (prop.)	**18.25**	**9.62**	**6.77**	**5.20**	**4.29**

B. Expressiveness and Computational Efficiency

One of the main advantages of VR-PRUNE over PRUNE is the increased expressiveness offered by variable token rates compared to binary (on/off) token rates of PRUNE. The increased expressiveness can be quantified in terms of graph size: with a more expressive model it is possible to represent the same functionality with fewer elements (actors, edges) than with a less expressive model. The *adaptive digital predistortion* and *parallel image classification* application examples that are introduced next highlight that in addition to expressiveness, variable token rates provide potential for saving power by computational effort reduction, while maintaining implementation efficiency.

1) Adaptive Digital Predistortion: Digital predistortion is a signal processing approach for compensating non-linear effects of a wireless transmitter's power amplifier [37]. The signal processing for predistortion is computationally very demanding, and therefore solutions [38] that trade-off predistortion bandwidth against computational effort are useful for saving power when possible, e.g. due to varying interference conditions.

Fig. 4 shows such an adaptive predistortion filter in two implementations: a) is a VR-PRUNE implementation, and b) is the conventional PRUNE implementation introduced in [28]. Both implementations describe the same 10-tap parallel Hammerstein filter structure [39], with adaptive functionality such that individual filter branches can be enabled or disabled on-the-fly at run time. In Fig. 4, s_1 and s_2 are source actors that provide samples from the transmitter baseband side; x is a dynamic actor that computes polynomial basis functions and distributes the samples to parallel FIR filter branches f; y is a dynamic actor that implements a summation function for combining the filter branch outputs and compensates for I/Q imbalance; t_1 and t_2 finally act as output actors towards the power amplifier. Adaptive processing is controlled by the configuration actor q that based on external input can enable/disable filter branches $f_1 \ldots f_4$ and $f_7 \ldots f_{10}$ in the PRUNE implementation. In the VR-PRUNE implementation the same effect is accomplished by variable token rates to/from the actor f. In the PRUNE implementation the FIR filter actors are SPAs, whereas the VR-PRUNE FIR actor f is a DPA.

It can be seen that the VR-PRUNE implementation (Fig. 4(a) compactly captures the adaptive predistortion functionality within 8 actors and 11 FIFOs, whereas the PRUNE implementation (Fig. 4(b) requires 17 actors and 46 FIFOs (double-line arrows stand for 2×FIFOs, one for the I channel, and one for the Q channel). This reduction of graph elements comes from the fact that VR-PRUNE is able to capture the functionality of actors $f_1 \ldots f_{10}$ within a single actor f, and consequently reduces the number of connecting FIFOs from 40 down to 4. Based on the number of architectural elements, VR-PRUNE therefore reduces the predistortion model complexity by 70% compared to PRUNE. This reduction ratio depends on the structure of the original dataflow graph, more specifically on the count of f (filter) actors.

One potential fallacy in evaluating model complexity reduction as described above is that actor and edge counts can often be reduced by just lumping actors together. For example, an entire SDF graph can be implemented as a single actor, thereby reducing the graph size to 1 actor and 0 edges. To show that this is not the case here, Table III shows the lines of code per actor for the Fig. 4 graphs. It can be seen, that on average, the VPRF implementation requires fewer lines per code than the PRUNE implementation. Looking at the dynamic actors x and y, it can be seen that the PRUNE implementations of these actors have around 30% more code than their VPRF counterparts, which is related to inter-actor communication: initiating and terminating inter-actor data transfer requires two lines of code per FIFO. Since the PRUNE x and y actors have 20 FIFOs towards the f actor each, whereas the VR-PRUNE x and y have only 2, the difference is obvious. In contrast, the VPRF f actor has 20 lines of code more than the PRUNE counterpart – this is due to the fact that the single f actor stores all the filter coefficients that in the PRUNE implementation are distributed over $f_1 \ldots f_{10}$.

Finally, Table IV shows the throughput of the adaptive digital predistortion application for both the i7 and N2 platforms. The throughput is largely limited by the compute resources of each platform, but the runtime framework's impact can still be seen in the performance figures: due to the considerably simpler graph structure, the VPRF implementation (Fig. 4(a) causes reduced computational overhead compared to the conventional PRUNE implementation (Fig. 4(b). It is worthwhile to point out that reducing the number of actors as between Fig. 4(b) and Fig. 4(a) evidently decreases potential for parallel execution. Therefore, if the underlying execution platform has spare cores, it is not advisable to collapse all actors $f_1 \ldots f_{10}$ into a single actor, but instead consider distributing the computation effort evenly over available compute resources.

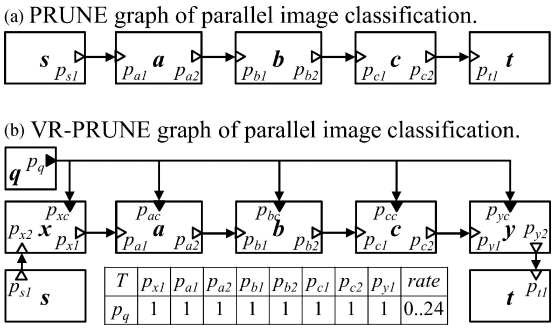


Fig. 5. The parallel image classification application. Subfigure (a) shows the static PRUNE graph, whereas (b) shows the adaptive VR-PRUNE graph. In both graphs, actors a, b, and c are executed on the GPU, and $N = 24$ images are classified in parallel.

2) Parallel Image Classification: Image classification is a fundamental computer vision task that is nowadays exclusively performed by deep CNNs. Driven for example by surveillance applications [40], there has also been interest in pushing classification to be done on low-resource computing devices. Some edge/IoT computing platforms such as the Odroid XU3 (Table II) are also equipped with a reasonably powerful GPU that offers optimal performance and efficiency only when the processing workload is sufficiently parallel.

Fig. 5(a) shows a PRUNE graph for parallel image classification: actor s is the source actor that acquires $N = 24$ parallel images from a source (e.g., camera interface), and using the underlying PRUNE CPU-GPU interface sends the N 96 $\times$ 96 RGB images in parallel to the GPU-mapped actor a that performs 2D convolution, ReLU non-linearity, max-pooling and $2\times$ subsampling to all N input images in parallel. Actor b (on GPU) also performs 2D convolution, max-pooling and subsampling, followed by the GPU-mapped actor c that implements a dense layer. After actor c, the feature maps of the N images are transferred back to CPU processing by the PRUNE infrastructure for final processing by two small dense layers, ReLU activations and softmax output, all performed by actor t. All actors in the Fig. 5(a) graph are SPAs.

Fig. 5(b) shows the VR-PRUNE graph for the same CNN-based image classifier, however modified such that the configuration actor q can at runtime determine which of the N parallel images will undergo classification, and which are skipped. For this, the VR-PRUNE graph contains additional dynamic actors x and y (a, b and c are DPAs in Fig. 5(b).

Fig. 6 shows a useful effect of this adaptivity: the computation effort decreases almost linearly on the i7 platform from the maximum of $N = 24$ processed images down to 0, offering great potential for saving processing power. Fig. 7 shows the similar effect on the XU3 embedded platform for $N = 4$ images.

Fig. 8 shows the power scaling effect of VPRF parallel image classification on the embedded XU3 platform. Summing the power dissipation of the GPU, CPU, and memory, it can be seen that when all frames (100%) are classified, the total power dissipation by the application is 3.4 W. Decreasing the number of classified frames down to 0% by adjusting the token rate, makes the application power dissipation as small as 20 mW. For each hardware component (GPU, CPU, memory), the power figures were acquired from the on-board current sensors of the XU3 platform.

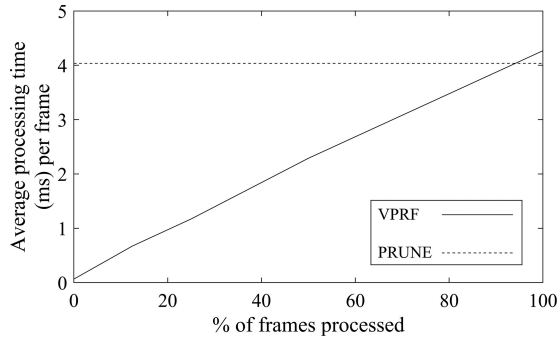


Fig. 6. Parallel image classification performance on the i7 GPU platform for $N = 24$ parallel images: static processing pipeline of PRUNE vs. VPRF adaptive processing by variable token rates.

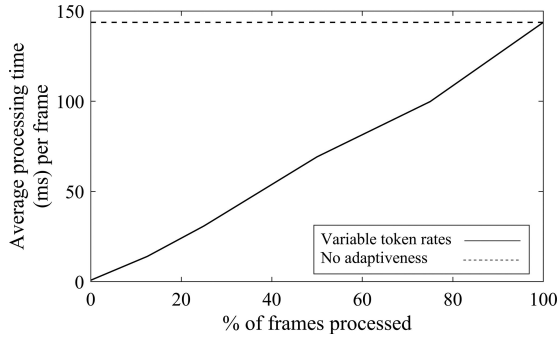


Fig. 7. Parallel image classification performance on the XU3 GPU platform for $N = 4$ parallel images. VPRF adaptive skipping of frames by variable token rates vs. no adaptiveness.

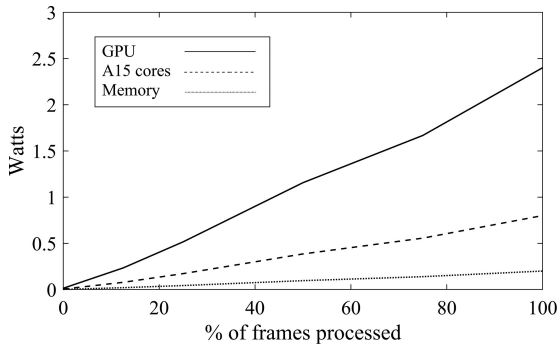
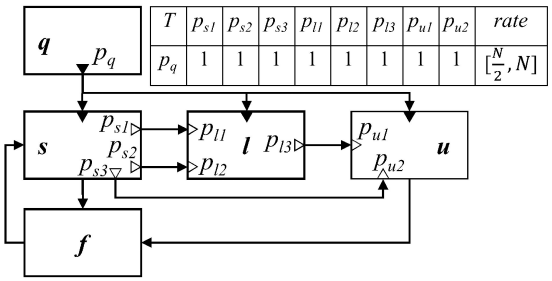


Fig. 8. Parallel image classification power dissipation on the XU3 platform under VPRF for $N = 4$ parallel images. The x axis depicts percentage of frames processed (e.g. 25% = on average every fourth frame undergoes classification).

(a) VR-PRUNE graph of the DU-DPD filter.



T	p_{s1}	p_{s2}	p_{s3}	p_{l1}	p_{l2}	p_{l3}	p_{u1}	p_{u2}	*rate*
p_q	1	1	1	1	1	1	1	1	$[\frac{N}{2}, N]$

(b) PRUNE graph of the DU-DPD filter.

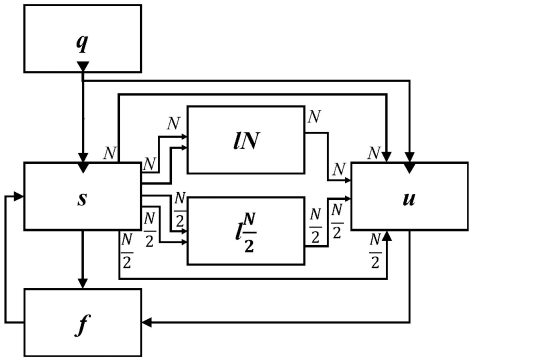


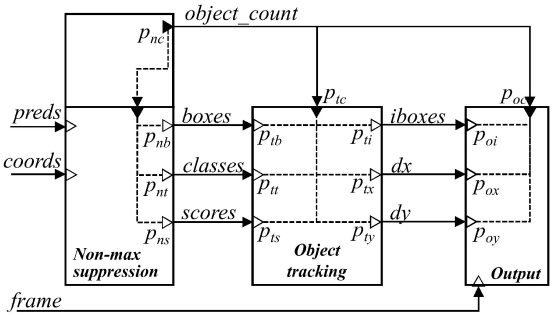
Fig. 9. The dynamic-update predistortion (DU-DPD) filter application. Subfigures: (a) VR-PRUNE and (b) PRUNE graphs. Both graphs implement two alternative token rates: N and $\frac{N}{2}$.

C. Further Application Examples

1) Dynamic-Update Predistortion Filter: The GPU-based dynamic-update predistortion filter (DU-DPD) for 5 G small cells is based on a recent architecture presented in [37]. The original dataflow implementation of [37] was imported to VPRF and modified such that the sample rate of the learning part can be changed adaptively at run-time for the purpose of saving computation effort in situations where radio frequency interference is low. The VR-PRUNE graph of the DU-DPD is shown in Fig. 9(a); the actor *s* contains essentially a power amplifier model, *l* encapsulates decorrelation-based filter coefficient learning functionality, whereas the actor *u* updates filter coefficients. Actual signal predistortion is performed by the *f* actor that has a constant I/O sample rate of N and is executed on the GPU. In practice, N can be e.g. 10000 or 65535 [37].

The variable-rate processing feature of the DU-DPD is exhibited between the actors *s*, *l* and *u:* the current sample rate (atr) can be adaptively changed among $\frac{N}{2}$ and N, as dictated by the *q* actor. The subset of the four aforementioned actors also form the sDPG for the DU-DPD application: *s* and *u* are the pair of dynamic actors δ for the sDPG, and *l* is a DPA. The actor *f* has static token rates at all its inputs and outputs, and hence it is an SPA.

With respect to computational characteristics, DU-DPD differs from the adaptive digital predistortion application of Section VI-B in two ways: 1) DU-DPD includes a feedback loop,



T	p_{nb}	p_{nt}	p_{ns}	p_{tb}	p_{tt}	p_{ts}	p_{ti}	p_{tx}	p_{ty}	p_{oi}	p_{ox}	p_{oy}	*rate*
p_{nc}	1	1	1	1	1	1	1	1	1	1	1	1	0..9

Fig. 10. The object detection and tracking application. VR-PRUNE graph of the SSD-Mobilenets object tracking component. The full application graph consists of 53 actors and 72 edges.

and 2) the learning algorithm (actor *l*) is recursive by nature. Here, recursiveness means that the algorithm operates on blocks of samples, and with respect to the algorithm's output, sample value s_{i+1} depends on sample value s_i ($i \in [0, N-1]$). Therefore, output sample s_{i+1} cannot be computed independently of sample s_i, for all $i \in [0, N-1]$.

The significantly higher expressiveness of VR-PRUNE over PRUNE is illustrated in Fig. 9(a) and Fig. 9(b). Using the PRUNE MoC, implementing several learning rates would require replicating the *l* actor for each different token rate; Fig. 9(b) shows an example, where the *l* actor can operate at sample rates N and $\frac{N}{2}$. For a finer-grained set of token rates of, e.g., 8 different sample rates, the *l* actor and its associated edges would need to be replicated 8 times. In contrast, the VR-PRUNE model is capable of supporting any number of integer sample rates with the simple graph that is shown in Fig. 9(a): the only change required is the token rate range (or rate list) in the control table.

2) Object Detection and Tracking: Visual object detection has been one of the most successful applications of computer vision since the introduction of deep CNNs. A CNN-based object detector can be understood to consist of a few main components: 1) A CNN-based feature extractor, 2) the actual object detector, and 3) object post-processing for application specific purposes. One of the most apparent post-processing operations for object detection is object tracking that can be used to discover the motion trajectories of objects and for maintaining object identifiers across sequences of images.

For VPRF, an object tracker was implemented based on the MobileNets [41] CNN for feature extraction, followed by the SSD (single-shot detector) [42] object detector, and an object tracking block.[3] Fig. 10 depicts the final stages of this 53-actor VR-PRUNE graph: the Non-max suppression (NMS) actor receives object predictions and coordinates from the feature detector CNN, resolving these into the number of detected objects (*object_count*). This number is distributed from the

[3]This MobileNet-SSD implementation is fully stand-alone and requires no external computer vision frameworks or libraries. The trained weights for CNN layers were extracted from TensorFlow using a prototype software tool written by one of the paper authors, M. Khan.

output control port p_{nc} to the Object tracking actor (DPA) and to the Output actor (DA) that performs visualization. The number of detected objects varies frame-by-frame between 0 and MAX_OBJECTS, which is a compile-time constant. Consequently, the number of detected objects directly controls how many bounding boxes (*boxes*), class predictions (*classes*) and prediction scores (*scores*) are communicated from NMS to the object tracking actor. Similarly, this number also adjusts the number of tracked boxes (*iboxes*), and motion vectors (*dx* and *dy*) between the Object tracking and Output actors. As Fig. 10 shows, the NMS actor is a joint configuration and dynamic actor, and hence the pair of dynamic actors $\delta(D)$ is {NMS, Output}. Such a merged actor can be beneficial for, e.g., simplifying the actor network [28]. For the purpose of consistency analysis, as described in Section V, the configuration and dynamic actor should be treated as separate actors, however.

Similar to the DU-DPD application example, implementing the varying object count feature of Fig. 10 without the variable token rate feature of VR-PRUNE would be complicated. The VR-PRUNE graph displayed in Fig. 10 shows an implementation where the tracked object count can vary between 0 and 9 (visible within the control table) — implementing this rate scalability feature within PRUNE would require 9 instances of the object tracking actor.

D. Comparison With Other Frameworks

For years, efficient computing has been of highest importance in signal processing and machine learning, especially in the embedded systems context. Unfortunately, achieving highest computation efficiency requires adapting software to the intricacies of the hardware architecture, which is a very work-intensive task. To this extent, computing hardware manufacturers active in the machine learning domain (NVidia, Intel, ARM, Qualcomm, etc.) have in the recent years released proprietary libraries and frameworks for accelerating machine learning inference on their computing architectures.

The performance of a dataflow flavored runtime framework consists of two components: 1) the actor implementations, and 2) inter-actor communication and synchronization. The scope of the proposed VPRF framework is entirely related to the latter (2) item, whereas towards actor implementations (1) VPRF is implementation-agnostic. To provide perspective over VPRF efficiency, this section shows some results for VPRF in the context of commercial frameworks that utilize optimized actor implementations (1).

In order to benchmark VPRF synchronization efficiency (2) meaningfully, we chose to adopt actor implementations (1) from ARMCL and oneDNN libraries for machine learning on ARM and Intel platforms, respectively. As the programming interface of ARMCL is not directly compatible with VPRF, the experiment required manual program adjustments to the VPRF implementation after automatic VPRF code generation.

Table V provides a performance evaluation on the Image classification application (Fig. 5(a), reflecting the performance of VPRF against TensorFlow, and the optimized actor implementations of ARMCL and oneDNN. The comparison between VPRF and PRUNE is omitted here, because with this graph the runtime computations for VPRF and PRUNE are identical. The first column of the table shows results of an experiment, where image classification has been performed by VPRF and TensorFlow[4] on the i7 CPU, both leveraging oneDNN. It can be seen that VPRF provides a substantially higher processing performance than TensorFlow, related to the fact that VPRF programs are compiled and optimized by the GNU C compiler, whereas TensorFlow emphasizes ease of programming by Python. The last row of column 1 shows for reference the execution time of the same program implemented as single-threaded C code, leveraging the same computation kernels as the VPRF implementation. VPRF distributes the computations across the different cores of the i7 platform, providing higher throughput than the Baseline implementation. For reference, the second row shows a performance figure from Fig. 6 where $N = 24$ images are classified in parallel on the i7's GPU using generic OpenCL actor implementations. It can be seen that the performance-optimized oneDNN CPU actor implementations outperform GPU acceleration in this case. Since TensorFlow requires a CUDA compatible GPU (and TensorFlow Lite requires Android or IOS for OpenCL), the column 2 experiment could not be done for the TensorFlow framework.

TABLE V
IMAGE CLASSIFICATION PERFORMANCE IN THE CONTEXT OF COMMERCIAL FRAMEWORKS. EACH CELL REFLECTS PROCESSING TIME IN MILLISECONDS PER FRAME FOR N IMAGES PROCESSED IN PARALLEL USING THE GRAPH OF FIG. 5(A), AND 100% OF IMAGES UNDERGO CLASSIFICATION

Framework	i7 CPU $N = 1$, oneDNN	i7 GPU $N = 24$	N2 CPU $N = 1$	N2 GPU ARMCL
VPRF	2.1	4.04	153.5	16.8
TensorFlow	13.9	n/a	236.9	n/a
Baseline C	2.9	n/a	n/a	16.6

The last two columns show results using the embedded N2 platform with and without use of the GPU. In the last column, actor implementations were adopted from the ARMCL library and executed on the GPU from a) the Baseline C language program, and b) VPRF, yielding almost identical performance. Since all the significant actors of the Image classification application are executed on the single GPU of the system, there are no possibilities to leverage concurrent computing, and consequently both the Baseline C and VPRF versions effectively execute the classification sequentially. *However, the result shows that the concurrent thread-based VPRF runtime adds negligible overhead compared to the ARMCL accelerated Baseline C implementation.* To achieve highest performance, *GPU-mapped FIFOs* were used in VPRF. Details on this technique are explained in Appendix A. Column 3 also shows a comparison between VPRF with generic actor implementations against TensorFlow[5] CPU on the N2 ARM platform, indicating faster execution for VPRF.

Finally, Table VI shows a detailed comparison of VR-PRUNE vs. other dataflow models in terms of graph sizes. The table shows that for each application VR-PRUNE clearly outperforms both PRUNE and SADF [19] models in expressiveness. VRDF [29] and Dataflow Process Networks (DPN) [21], on the other hand, are more expressive than VR-PRUNE and can consequently capture VR-PRUNE graphs with similar, but a slightly fewer number of components. However, the DPN model

[4]Python 3.6.9, TensorFlow 2.3.1 with eager execution and JiT compilation enabled.

[5]Python 3.5.7, TensorFlow 1.14

TABLE VI
GRAPH SIZE VERSUS OTHER DATAFLOW FRAMEWORKS. THE *RATE* ROW EXPRESSES TOKEN RATE VARIATION FOR EACH APPLICATION, E.G. [0..24] EQUALS TO 25 DIFFERENT TOKEN RATES, AND "$\frac{N}{2}$ OR N" EXPRESSES TWO ALTERNATIVE RATES. FOR EACH FRAMEWORK AND APPLICATION, THE CELLS DENOTE NUMBER-OF-VERTICES, NUMBER-OF-EDGES. FOR EXAMPLE: (8, 11): 8 VERTICES AND 11 EDGES

Application	Adaptive DPD	Image cl.	DU-DPD	Object det.
Rate	[1..5]	[0..24]	$\frac{N}{2}$ or N	[0..9]
Figure	Fig. 4	Fig. 5b)	Fig. 9	Fig. 10
VR-PRUNE	8, 11	8, 11	5, 10	53, 72
PRUNE	17, 46	77, 100	6, 13	61, 119
SADF	60, 140	1000, 1250	9, 18	565, 920
VRDF/DPN	7, 8	7, 6	4, 7	53, 70

is so general that possibilities for graph consistency verification at design time are very limited, whereas for VRDF no practical design frameworks have been released so that the MoC's applicability to high processing performance could be evaluated. A detailed explanation of the graph size calculations is presented in Appendix B.

VII. DISCUSSION AND FUTURE WORK

Section I presented a weakly consistent dataflow graph that was originally introduced in [1], with a note that for example the TensorFlow dataflow environment produces version-dependent behavior upon execution of this graph.

How does VR-PRUNE handle this graph of Fig. 1? As such, the graph is not a valid VR-PRUNE graph. Since the graph contains dynamic token rate ports, an sDPG structure needs to be identified for VR-PRUNE compliance. Evidently, *conf* would serve as the configuration actor q, whereas *switch* and *proc* would represent the pair of dynamic actors δ associated with q. However, VR-PRUNE requires that both dynamic actors of δ need to be controlled by the same configuration actor, and hence the output of *conf* is required to be connected to *proc* as well. Now, the output port p_{sT} of *switch* and its counterpart p_{p1} in *proc* could be interpreted as DRPs controlled by the output of *conf*. With these changes, the graph would be a valid VR-PRUNE graph, and would also avoid unbounded use of memory: upon emitting a *False* token, the *conf* actor would set the atrs of the two DRPs (p_{sT} and p_{p1}) to zero, allowing *proc* to consume the token arriving from *input* independent of sample values. This example highlights the importance of formal design rules and computation models for detecting and diagnosing model flaws as early as possible.

Since one of the major attributes of VPRF is high processing performance on multicore and heterogeneous (CPU+GPU) platforms, one might wonder how VPRF compares to other similar frameworks in terms of performance. Table IV and Fig. 6 illustrate the processing performance of VPRF versus PRUNE, showing that performance differences between the frameworks are minimal, which is expectable because the differences in the runtime frameworks are modest. On the other hand, the PRUNE paper [28] presented extensive benchmarks, where it was shown that PRUNE outperformed the DAL [35] framework in all application benchmarks.

Currently, VPRF only supports a single OpenCL device in the system. As future work, this could be extended to enable multiple OpenCL devices including multiple GPUs and/or OpenCL-compatible CPUs. A further interesting direction worth exploring would be introducing distributed computing [35] for executing parts of VR-PRUNE graphs in a cloud similar to [43].

VIII. CONCLUSION

In this paper we have presented VR-PRUNE, a Model of Computation for high-performance signal processing applications, which features variable token rates and is accompanied with VPRF, a runtime library that has deeply integrated support for heterogeneous computing. VPRF is going to be released as open source similar to its predecessor PRUNE.

We have formally defined the VR-PRUNE Model of Computation, design rules, and consistency analysis, and have discussed its decidability. Compared to previous related Models of Computation, VR-PRUNE offers a unique combination of analyzability, expressiveness and practical applicability for high-performance applications.

Through extensive experiments using VPRF with four application examples, we have shown how VR-PRUNE

- Is applicable to practical signal processing algorithms,
- Offers considerably higher expressiveness than previous work,
- Enables adaptive processing for saving power, and
- Provides high processing performance.

APPENDIX A
GPU-MAPPED FIFOS

Heterogeneous computing across CPU and GPU resources needs to be implemented carefully to avoid unnecessary computation time overheads. One significant source of overhead in GPU based computation acceleration are memory transfers between the CPU and the GPU.

In VPRF, the memory structures related to dataflow actors and FIFOs reside by default in CPU memory. However, especially in machine learning applications, it is common that the application consists of a pipeline of actors (neural network layers) that are processed on the GPU. In such cases, highest performance is achieved when FIFOs between GPU-mapped actors reside in the GPU memory, avoiding unnecessary data transfers between the CPU and GPU: tokens flow within GPU memory from one GPU-mapped actor to the next GPU-mapped actor.

Although it is common practice in GPU computing to maintain intermediate data between GPU kernels ($\approx$ actors) in GPU memory buffers ($\approx$ FIFOs), implementing GPU buffering in the dataflow computing context requires some additional consideration to maintain data-driven application behavior.

In VPRF, this is achieved such that each FIFO buffer, which is mapped to GPU memory, has a dummy counterpart in CPU memory. This dummy counterpart of a FIFO does not carry any data (as the token data is in GPU memory) — it only serves for implementing synchronization between actors. By using these dummy *synchronization FIFOs*, the VR-PRUNE application simultaneously maintains data-driven behavior, while avoiding computation time overhead by keeping token data in GPU memory.

In the Image classification application, the use of GPU-mapped FIFOs decreases average image classification time from

18.7 ms to 16.2 ms when the ARMCL library is used for actor implementations on the N2 platform.

APPENDIX B
GRAPH COMPLEXITY COMPARISON

Details of the graph complexity comparison in terms of edge counts and vertex counts are explained below for PRUNE, SADF [19] and VRDF [29] models.

In terms of graph complexity, the PRUNE [28] model differs from VR-PRUNE in two significant ways: 1) PRUNE does not inherently support variable token rates, and hence token rate changes need to be emulated by a series of actor (vertex) instances that can individually be enabled or disabled, which increases graph component count compared to VR-PRUNE. 2) PRUNE does not require edges from configuration actors to those actors that are enabled/disabled at runtime. This decreases graph component count compared to VR-PRUNE. For an illustration related to these differences, the reader should refer to Fig. 4 whose subfigures a) and b) show equivalent PRUNE and VR-PRUNE graphs. Related to the PRUNE model of the Fig. 5 parallel image classifier, it is important to notice that subfigures a) and b) do *not* depict graphs of equivalent behavior. A PRUNE graph equivalent to the Fig. 5(b) VR-PRUNE graph would consist of 24 instances of actors a, b, and c, which would result in the 77 actors and 100 edges, as shown in Table VI.

The SADF model captures dynamic (runtime) changes on graph topology by scenarios such that each possible topology is described by a separate SDF graph. In the Adaptive DPD application the number of active filter branches ranges between 1 and 5, resulting in 5 scenarios (graphs), each of which has a different number of actors and edges in the branches: $S^{adpd}_{edges} = \Sigma^{N}_{i=1} 8i + 4$, and $S^{adpd}_{vertices} = \Sigma^{N}_{i=1} 2i + 6$. For $N = 5$, a total of 60 actors and 140 edges ensue when all scenarios are added together. In the Image classification application, between 0 and $N = 24$ images can be classified in parallel, resulting in 25 graph scenarios: $S^{imcl}_{edges} = \Sigma^{N}_{i=0} 4i + 2$, and $S^{imcl}_{vertices} = \Sigma^{N}_{i=0} 3i + 4$, which totals into 1000 actors and 1250 edges (the numbers are exact) for $N = 24 + 1$. For DU-DPD the number of scenarios is $N = 2$ ($S^{dudpd}_{edges} = \Sigma^{N}_{i=1} 4i + 3$, and $S^{dudpd}_{vertices} = \Sigma^{N}_{i=1} i + 3$), and for Object detection there are $N + 1 = 10$ scenarios: $S^{objd}_{edges} = \Sigma^{N}_{i=0} 6i + 65$, and $S^{objd}_{vertices} = \Sigma^{N}_{i=0} i + 52$. Since SADF does not explicitly mention the need of configuration actors, the component counts do not include the configuration actor, or edges connected to the configuration actor.

Finally, the VRDF [29] and DPN [21] models are slightly more expressive than VR-PRUNE: neither of these models requires symmetric token rates, nor any need for configuration actors. To this extent both VRDF and DPN are able to directly capture VR-PRUNE graphs, albeit without configuration actors and edges. Consequently VRDF and DPN graph complexities are very similar, but slightly lower than those of VR-PRUNE graphs.

REFERENCES

[1] J. T. Buck, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," Ph.D. dissertation, Dept. Elect. Eng. and Comput. Eng., Univ. California Berkeley, 1993.

[2] G. F. Zaki, W. Plishker, S. S. Bhattacharyya, C. Clancy, and J. Kuykendall, "Integration of dataflow-based heterogeneous multiprocessor scheduling techniques in GNU radio," *J. Signal Process. Syst.*, vol. 70, no. 2, pp. 177–191, 2013.

[3] M. Abadi *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2016, *arXiv:1603.04467*.

[4] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.

[5] K. Ravindran, A. Ghosal, R. Limaye, G. Wang, G. Yang, and H. Andrade, "Analysis techniques for static dataflow models with access patterns," in *Proc. Conf. Des. Architectures Signal Image Process.*, 2012, pp. 1–8.

[6] *Agilent EEsof EDA SystemVue 2011 Technical Overview*, Agilent Technologies, Inc., May 2011.

[7] A. Veit and S. Belongie, "Convolutional networks with adaptive inference graphs," in *Proc. Eur. Conf. Comput. Vis.*, 2018, pp. 3–18.

[8] R. T. Mullapudi, W. R. Mark, N. Shazeer, and K. Fatahalian, "Hydranets: Specialized dynamic architectures for efficient inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 8080–8089.

[9] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, 1993, vol. 1, pp. 429–432.

[10] Y. Yu *et al.*, "Dynamic control flow in large-scale machine learning," in *Proc. 13th EuroSys Conf.*, 2018, pp. 1–15.

[11] Y. Ma, J. Wu, S. S. Bhattacharyya, and J. Boutellier, "Decidable variable-rate dataflow for heterogeneous signal processing systems," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, 2020, pp. 1683–1687.

[12] J. L. Peterson, "Petri nets," *ACM Comput. Surv.*, vol. 9, no. 3, pp. 223–252, 1977.

[13] K. M. Kavi, B. P. Buckles, and U. N. Bhat, "A formal definition of data flow graph models," *IEEE Trans. Comput.*, vol. 35, no. 11, pp. 940–948, Nov. 1986.

[14] K. M. Kavi, B. P. Buckles, and U. N. Bhat, "Isomorphisms between Petri nets and dataflow graphs," *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 10, pp. 1127–1134, Oct. 1987.

[15] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "PREESM: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming," in *Proc. Eur. Embedded Des. Educ. Res. Conf.*, 2014, pp. 36–40.

[16] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. Int. Conf. Compiler Construction*, Springer, 2002, pp. 179–196.

[17] H. P. Huynh, A. Hagiescu, O. Z. Liang, W.-F. Wong, and R. S. M. Goh, "Mapping streaming applications onto GPU systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 9, pp. 2374–2385, Sep. 2014.

[18] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *IEEE Trans. Signal Process.*, vol. 44, no. 2, pp. 397–408, Feb. 1996.

[19] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *Proc. Int. Conf. Embedded Comput. Syst.*, 2011, pp. 404–411.

[20] E. Jeong, D. Jeong, and S. Ha, "Dataflow model-based software synthesis framework for parallel and distributed embedded systems," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 26, no. 5, pp. 1–38, 2021.

[21] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[22] J. Eker and J. W. Janneck, "CAL language report," UC Berkeley, Tech. Rep. UCB/ERL M03/48, 2003.

[23] M. Mattavelli, I. Amer, and M. Raulet, "The reconfigurable video coding standard [standards in a nutshell]," *IEEE Signal Process. Mag.*, vol. 27, no. 3, pp. 159–167, May 2010.

[24] G. Cedersjö and J. W. Janneck, "Tÿcho: A framework for compiling stream programs," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 6, pp. 1–25, 2019.

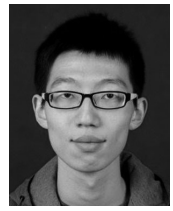
[25] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "ORCC: Multimedia development made easy," in *Proc. ACM Int. Conf. Multimedia*, 2013, pp. 863–866.

[26] O. Rafique and K. Schneider, "SHeD: A framework for automatic software synthesis of heterogeneous dataflow process networks," in *Proc. Euromicro Conf. Digit. Syst. Des.*, 2020, pp. 1–10.

[27] G. Gao, R. Govindarajan, and P. Panangaden, "Well-behaved dataflow programs for DSP computation," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, 1992, vol. 5, pp. 561–564.
[28] J. Boutellier, J. Wu, H. Huttunen, and S. S. Bhattacharyya, "PRUNE: Dynamic and decidable dataflow for signal processing on heterogeneous platforms," *IEEE Trans. Signal Process.*, vol. 66, no. 3, pp. 654–665, Feb. 2018.
[29] M. H. Wiggers, M. J. Bekooij, and G. J. Smit, "Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2008, pp. 183–194.
[30] L. Schor, A. Tretter, T. Scherer, and L. Thiele, "Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL," in *Proc. IEEE Symp. Embedded Syst. Real-time Multimedia*, 2013, pp. 41–50.
[31] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.
[32] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "A hierarchical multiprocessor scheduling system for DSP applications," in *Proc. Asilomar Conf. Signals, Syst. Comput.*, 1996, vol. 1, pp. 122–126.
[33] J. McAllister and M. Davis, "Graph coordination for compact representation of regular dataflow structures," in *Proc. IEEE Workshop Signal Process. Syst.*, 2020, pp. 1–6.
[34] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real-time DSP," in *Proc. IEEE Glob. Telecommun. Conf. Exhibition'Commun. Technol. 1990 s Beyond*', 1989, pp. 1279–1283.
[35] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele, "Scenario-based design flow for mapping streaming applications onto on-chip many-core systems," in *Proc. Int. Conf. Compilers, Architectures Synth. Embedded Syst.*, 2012, pp. 71–80.
[36] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing With openCL: Revised openCL 1 1.2 edition* Newnes, 2012.
[37] P. P. Campo, V. Lampu, A. Meirhaeghe, J. Boutellier, L. Anttila, and M. Valkama, "Digital predistortion for 5G small cell: GPU implementation and RF measurements," *J. Signal Process. Syst.*, vol. 92, no. 5, pp. 475–486, 2020.
[38] M. Aghababaeetafreshi, D. Korpi, M. Koskela, P. Jääskeläinen, M. Valkama, and J. Takala, "Software defined radio implementation of a digital self-interference cancellation method for inband full-duplex radio using mobile processors," *J. Signal Process. Syst.*, vol. 90, no. 10, pp. 1297–1309, 2018.
[39] L. Anttila, P. Handel, and M. Valkama, "Joint mitigation of power amplifier and I/Q modulator impairments in broadband direct-conversion transmitters," *IEEE Trans. Microw. Theory Techn.*, vol. 58, no. 4, pp. 730–739, Apr. 2010.
[40] K. Muhammad, S. Khan, V. Palade, I. Mehmood, and V. H. C. De Albuquerque, "Edge intelligence-assisted smoke detection in foggy surveillance environments," *IEEE Trans. Ind. Informat.*, vol. 16, no. 2, pp. 1067–1075, Feb. 2020.
[41] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.
[42] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Proc. Eur. Conf. Comput. Vis.*, Cham, Switzerland: Springer, 2016, pp. 21–37.
[43] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Comput. Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

Jani Boutellier (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees from the University of Oulu, Oulu, Finland, in 2005 and 2009, respectively. He is currently an Associate Professor with the School of Technology and Innovations, University of Vaasa, Vaasa, Finland. In 2007-2008, 2013 he was a Visiting Researcher with the Swiss Federal Institute of Technology Lausanne, Lausanne, Switzerland. His research interests include dataflow programming, signal processing, heterogeneous computing and machine learning for computer vision. Between 2016 and 2021, he was a Member of the IEEE Signal Processing Society Design and Implementation of Signal Processing Systems (DISPS/ASPS) Technical Committee.

Yujunrong Ma received the bachelor's degree in automation from the Harbin Institute of Technology, Harbin, China, the Ph.D. degree from the Department of Electrical and Computer Engineering, University of Maryland, University of Maryland, College Park, MD, USA. He held third position with the National Institute of Justice (NIJ) recidivism prediction challenge. His research interests include dataflow implementations, deep learning and evolutionary algorithms. He was the recipient of the Dean's Fellowship in 2019.



Jiahao Wu received the bachelor's degree from the University of Electronic Science and Technology of China, Chengdu, China, the Ph.D. degree from the Department of Electrical and Computer Engineering, University of Maryland, College Park, MD, USA. His research interests include model-based design for parallel computing, dataflow implementations and synthesis of digital signal processing systems.

Mir Khan received the bachelor's degree from the University College of Bahrain, Janabiyah, Bahrain, the master's degree from Tampere University, Tampere, Finland, where he is currently working toward the Ph.D. degree. His research interests include optimizing neural networks inference implementations for graphical processing units and embedded systems.

Shuvra S. Bhattacharyya (Fellow, IEEE) received the Ph.D. degree from the University of California, Berkeley, Berkeley, CA, USA. He is currently a Professor with the Department of Electrical and Computer Engineering, University of Maryland, College Park, MD, USA. He holds a joint appointment with the University of Maryland Institute for Advanced Computer Studies, and is affiliated with the Maryland Crime Research and Innovation Center. He also holds a part time visiting position as the Chair of Excellence in Design Methodologies and Tools with the Institut National Des Sciences Appliquées, Rennes, France. He has held industrial positions as a Researcher with the Hitachi America Semiconductor Research Laboratory, and Compiler Developer at Kuck & Associates. From 2015 to 2018, he was a part time Visiting Professor with the Department of Pervasive Computing, Tampere University of Technology (now Tampere University), Tampere, Finland, as part of the Finland Distinguished Professor Programme. He has also held Visiting Research positions with the U.S. Air Force Research Laboratory.