Resource-Constrained Implementation and Optimization of a Deep Neural Network for Vehicle Classification

Renjie Xie*, Heikki Huttunen*, Shuoxin Lin†, Shuvra S. Bhattacharyya*†, Jarmo Takala*
* *Department of Pervasive Computing*
Tampere University of Technology, Finland
† *Department of Electrical and Computer Engineering*
University of Maryland, College Park, USA
Email: {renjie.xie, heikki.huttunen}@tut.fi, {slin07, ssb}@umd.edu, jarmo.takala@tut.fi

Abstract—**Deep learning has attracted great research interest in recent years in many signal processing application areas. However, investigation of deep learning implementations in highly resource-constrained contexts has been relatively unexplored due to the large computational requirements involved. In this paper, we investigate the implementation of a deep learning application for vehicle classification on multicore platforms with limited numbers of available processor cores. We apply model-based design methods based on signal processing oriented dataflow models of computation, and using the resulting dataflow representations, we apply various design optimizations to derive efficient implementations on three different multicore platforms. Using model-based design techniques throughout the design process, we demonstrate the ability to flexibly experiment with optimizing design transformations, and alternative multicore target platforms to achieve efficient implementations that are tailored to the resource constraints of these platforms.**

Keywords-**Dataflow, deep learning, model-based design, multicore platforms, signal processing systems.**

I. Introduction

The proliferation of research on deep learning applications and concurrent advances in application areas for ubiquitous embedded computing, such as automotive embedded systems and the Internet of things, motivate the investigation of design methodologies for deploying deep neural network (DNN) systems on resource-constrained embedded platforms. Based on this motivation, we investigate in this paper the resource-constrained implementation of deep learning applications, using vehicle classification as a concrete case study throughout our investigation. As a core part of the design methodology developed in this paper, we apply model-based design methods based on signal processing oriented dataflow models of computation, and we employ the resulting dataflow representations to implement, experiment with, and iteratively optimize deep learning vehicle classification on three different multicore platforms using limited numbers of processing cores.

More specifically, this paper introduces a unified methodology for modeling, mapping, and transforming deep-learning implementations using dataflow techniques, along with methods to integrate the hyperparameter tuning and simulation processes of deep learning system design with the proposed dataflow-based implementation approach. While this methodology is not specific to any particular application area, it is particularly well suited to embedded signal, image, and video processing applications, where dataflow-based design is especially relevant (e.g., see [1]). As mentioned above, we employ vehicle classification as a case study to concretely demonstrate the methodology throughout the paper.

II. Related Work

Various designs for deep neural networks have been proposed in the literature (e.g., see [2], [3], [4], [5]). Our work presented in this paper is complementary to these works as it develops methods for modeling and mapping different network architectures into efficient embedded implementations.

TensorFlow is a recently-introduced design framework for large-scale machine learning systems, including systems that employ DNNs [6]. While TensorFlow applies dataflow modeling principles as we do in our work, our work is distinguished by its emphasis on resource-constrained implementation and on flexible experimentation with alternative dataflow modeling, scheduling, and transformation techniques, which are important for exploring the complex design spaces involved in embedded DNN implementation.

In contrast to TensorFlow, which applies specialized dataflow modeling and scheduling techniques, our proposed design methodology promotes experimentation with dataflow modeling and scheduling aspects as an integral part of the design process. Such experimentation is promoted by our use of the lightweight dataflow environment (LIDE) [7], which we describe further in Section IV. Additionally, our emphasis on formulating the DNN design process in terms of abstract dataflow principles and associated scheduling techniques helps to promote future integration of methods developed in our work with arbitrary dataflow-based design tools for signal processing system design (e.g., see [1] for coverage of this diverse and important class of tools), including — but not limited to — TensorFlow.

III. DNN Topology for Vehicle Classifier

The deep learning application that we focus on in this paper is that of image-based recognition of vehicles. In particular, we develop DNN implementations for automatic discrimination among four types of vehicles — bus, truck, van and car. For this application, we build on the DNN network structure derived in recent work on DNN-based vehicle classification [8], and we go beyond this previous work by investigating aspects related to the efficient embedded implementation of this structure.

Using Caffe [9], we apply random search to optimize the selection of hyperparameters. After a series of search iterations using 50 sets of randomly-generated hyper-parameters, we derive the optimized set of hyperparameters for our vehicle classification application.

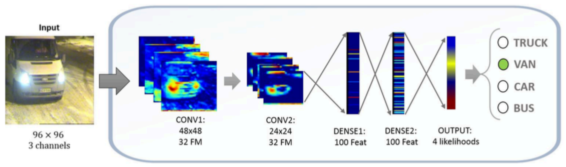

Figure 1. The structure of the proposed network.

The DNN architecture (Figure 1) is composed of five layers — two convolutional layers, followed by two dense layers, and finally, the classifier layer. The first layer consists of three channels — corresponding to red, green and blue (RGB) color channels — of the input image that are each convolved into 32 feature maps and then maxpooled to 48×48 resolution. Here, each input channel has dimensions 96×96. In the second layer, the sets of 32 feature maps are convolved again and then downsampled to 24×24 with maxpooling. The third and fourth layers are two fully connected layers with 100 nodes each. The output layer performs a mapping from 100 features to 4 class likelihoods using a softmax operator. Between pairs of adjacent layers, Rectified Linear Unit (ReLU) nonlinearity is applied.

For more details on the underlying DNN configuration from which this system is derived, we refer the reader to [8].

IV. Lightweight Dataflow Design and Implementation

After deriving the hyperparameters as described in Section III, we implement the DNN system in MATLAB for simulation and testing purposes. A primary objective of this step in connection with our overall embedded system design process is to collect results from each layer so that the embedded implementation for each layer can be tested in isolation in addition to performing complete, system-level tests of the target implementation. Such layer-by-layer testing helps to build up the implementation incrementally, and localize the causes of test failures to provide for more rapid design iterations.

After developing the MATLAB-based simulation model for our DNN-based vehicle classification system, we proceed to develop an initial dataflow-based implementation, which will be employed as a starting point to evaluate the system on candidate multicore platforms, and iterate on the design through optimizing dataflow graph transformations that improve implementation performance.

To develop and iteratively optimize our dataflow-based implementation, we employ the lightweight dataflow environment (LIDE), which is a dataflow-based programming environment that allows signal processing system designers to apply and experiment with dataflow modeling approaches relatively quickly and flexibly in the context of existing design processes [7], [10]. In particular, we employ LIDE-C, which is a part of the LIDE environment that is designed for use with C as the language for implementing dataflow-based software components (*actors*). LIDE-C provides application programming interfaces (APIs) that can be used when developing software modules using C such that the modules can be integrated together systematically as actors in an enclosing dataflow graph. This allows complete signal processing systems, such as our targeted DNN-based vehicle classification system, to be constructed as dataflow-based signal flow graph implementations where the actors are realized in C. Our use of LIDE-C in this work, as compared to other variants of LIDE, is motivated by the important role of C in embedded software implementation. For more details on LIDE, we refer the reader to [7], [10].

In LIDE, as in other related dataflow environments, a signal processing application is represented as a directed graph in which the vertices (actors) represent computational tasks, and each edge corresponds to a first-in, first-out (FIFO) communication channel that buffers data as it passes from the output of one actor to the input of another.

As a first step in formulating the overall dataflow model for the DNN system, we convert the diagrammatic sketch of Figure 1 into the block diagram shown in Figure 2.

Although this block diagram, when its hierarchical representation is flattened, encompasses thousands of individual

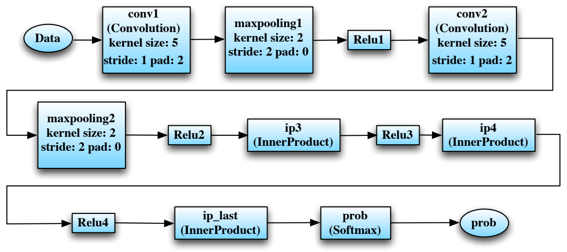
Data
conv1 (Convolution) kernel size: 5 stride: 1 pad: 2
maxpooling1 kernel size: 2 stride: 2 pad: 0
Relu1
conv2 (Convolution) kernel size: 5 stride: 1 pad: 2
maxpooling2 kernel size: 2 stride: 2 pad: 0
Relu2
ip3 (InnerProduct)
Relu3
ip4 (InnerProduct)
Relu4
ip_last (InnerProduct)
prob (Softmax)
prob

Figure 2. Block diagram representation of the DNN system.

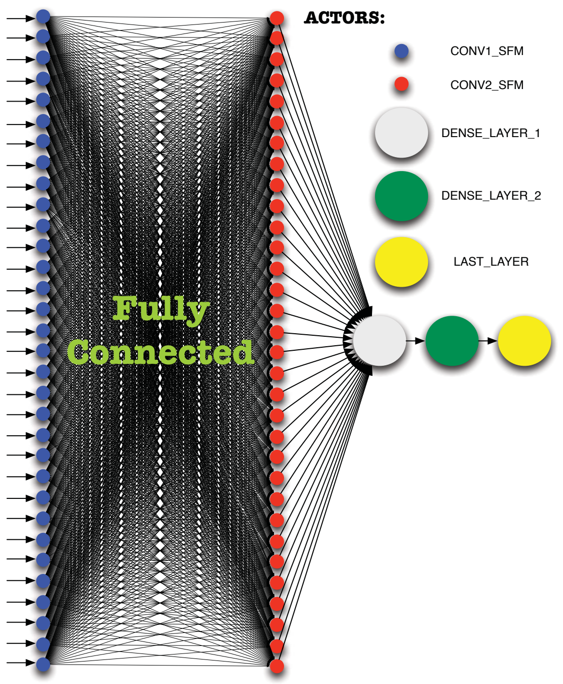

Figure 3. Top-level dataflow model.

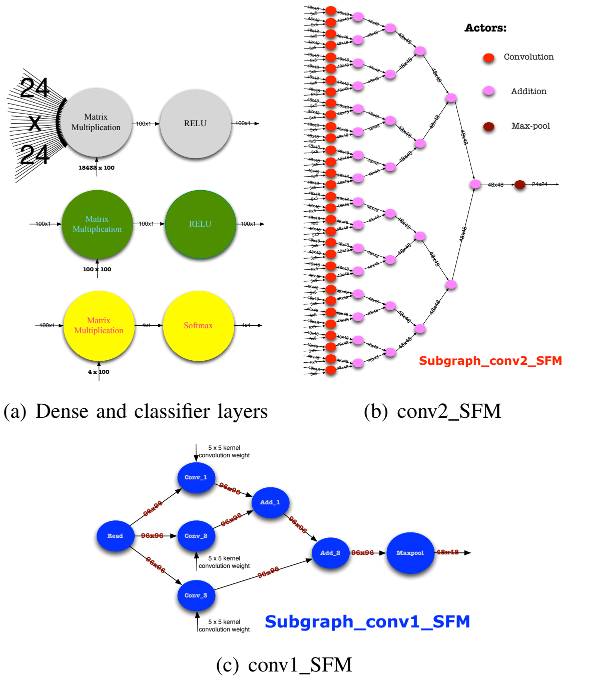

Figure 4. Internal (nested) dataflow representations for different types of hierarchical actors.

signal processing blocks (actors), there is a great deal of regularity in the way the blocks are instantiated and connected. Such regularity can be exploited in deriving LIDE-C designs in the form of compact, parameterized dataflow graph implementations that designers can efficiently analyze and manipulate (e.g., see [11]). The block diagram in Figure 2 incorporates a total of 10 different types of actors, which are summarized as follows.

• Read Channel: this actor decomposes an input image stream into separate RGB channels, where each channel carries 96×96 matrices as its basic type.

• Convolutional Actor: this actor performs a convolution on two arrays of inputs, which arrive through two input FIFOs, and outputs an array of the same size and dimensionality as the inputs.

• Maxpool: this actor performs a form of nonlinear down-sampling, where the input image is partitioned into sub-regions of non-overlapping rectangles, and the maximum value for each such sub-region is produced on the actor output.

• ReLU: This actor performs the function $f(x) = max(0, x)$ to derive an output value from each input value x. This is an important nonlinear function that is applied in DNNs.

• Softmax: This actor performs the operation $a = exp(n)/sum(exp(n))$ across the net inputs of a given layer.

• Write Actor: writes data to a given text file.

• All_to_one : This actor concatenates multiple smaller input matrices into a larger output matrix (e.g., three 3×3 matrices combined into a single 9×3 matrix).

• Broadcast Actor: makes copies onto multiple output ports of matrices that are received on its single input port.

• Multi_mtx_add Actor: This actor performs elementwise addition across multiple input matrices to produce a single “sum matrix”.

• Matrix Multiplication.

Figure 3 shows the dataflow model that is constructed through the process of transforming abstract actors from the block diagram representation (Figure 2) into concrete, LIDE-C-based actor implementations. All of the actors shown in Figure 3 are hierarchical actors; internal functionalities for these hierarchical actors are shown in Figure 4.

V. DNN Dataflow Graph Transformations

In this section, we describe a selected subset of the transformations that we apply on our LIDE-C-based DNN implementation to improve its performance on the targeted multicore platforms. These transformations exploit the orthogonalization of actor implementation, task scheduling, and buffer management in LIDE, which allow for rapid prototyping of alternative implementation strategies for the given dataflow graph [10]. While these transformations are not new design optimizations in and of themselves (e.g., see [12], [1], their integration into resource-constrained multicore DNN implementations, and their application based on lightweight dataflow design principles are novels aspect of this work.

Broadcast optimization. In this optimization, we transform the copying of data values associated with each broadcast actor into in-place read and write operations on a single buffer, where all consuming actors read from this single, shared buffer. This eliminates large numbers of data copying operations that result from conventional dataflow-based implementations of the broadcast actors.

Simplify connections in the first two layers. By adding three actors — two all_to_one actors and one broadcast actor — between the first two convolutional layers, we are able to decrease the complexity of the dataflow and improve the efficiency of the connections between these layers.

In-place operations for images and matrices. Instead of loading images from actor input FIFOs into internal storage for the actors, we utilize image data directly from the input FIFOs. Similarly, groups of matrix additions are performed in place on actor input FIFOs using the multi_mtx_add actor described in Section IV, which avoids data transfers between actors when the additions are performed using groups of smaller-scale matrix addition actors.

Clustering into threads. In this transformation, we cluster (or group together) subgraphs within the overall DNN dataflow graph to be executed as concurrent threads in the target multicore implementation. This enables parallel execution of DNN subsystems when multiple cores are employed. Execution within the subgraph for each thread is managed by a LIDE-C-based dataflow graph scheduler that is dedicated to the thread, and the different schedulers for the different threads therefore execute concurrently for the overall DNN system. We employ pthreads as the interface for implementing the thread-based concurrent execution of the dataflow subgraph schedules [13]. In our experimentation with alternative clusterings, we find that parallelization of feature map computations is especially effective in improving performance on the target platforms.

VI. Functional Validation

Functional validation is a critical step before applying design transformations and for automatically validating the correctness of each implementation iteration as different transformations are applied. For this purpose, we apply the DSPCAD Integrative Command Line Environment (DICE), which provides language- and platform-agnostic features for testing of embedded signal processing software [14], [15].

Figure 5 gives an illustration of the DICE-based organization and associated directory hierarchy of the software and test modules for the DNN system design. For further details on development and testing of signal processing systems using DICE, we refer the reader to [15].

VII. Experimental Results

Figure I shows the processing times measured from executing three versions of our DNN system for vehicle classification: (1) simulation implementation in MATLAB;

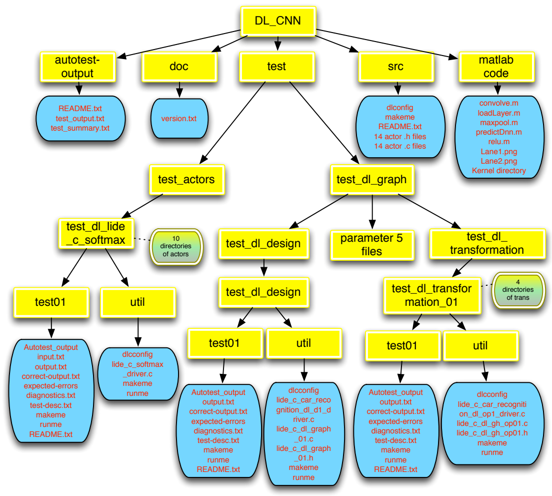

Figure 5. DICE-based design organization of the DNN system.

(2) unoptimized LIDE-C implementation; and (3) LIDE-C implementation with the optimization techniques discussed in Section IV and Section V. Both versions (2) and (3) include the effects of turning on C compiler optimizations, but differ in whether or not the higher level, dataflow transformations described in Section IV and Section V are applied.

Here, measurements are shown both for t_L and t_C, where t_L is the time to load the DNN parameters obtained from training, and t_C is the time required for classification of a single input image once the training parameters have been loaded. Since the cost of loading data is incurred only once, at system setup time, the value of t_C is generally more important for our design context. The results reported in Figure I are derived using a single-core, Intel Core i5-4248U processor with 8 GB memory. The results indicate that applying model-based optimization techniques in LIDE-C significantly improves the overall single-core performance compared to both the simulation version and the unoptimized (initial) LIDE-C version, while also exposing high-level dataflow structure that can be exploited to map the application onto multicore configurations.

Time	Matlab	LIDE-C (not optimized)	LIDE-C (optimized)
$t_L(sec)$	76.7	1.6	1.5
$t_C(sec)$	3.6	44.8	2.7

Table I
Single-core processing times for three design versions.

With the aid of the clustering transformation described in Section V, we then map the optimized LIDE-C implementation to parallelize the design for efficient execution on various alternative multicore platform configurations.

The results in terms of classification time performance are summarized in Figure II. These results quantitatively demonstrate the utility of optimized LIDE-C implementations in exploring complex design spaces for DNN systems involving alternative multicore platforms, and in achieving further performance improvement by exploiting parallelism in these platforms.

Platforms	Number of Cores	$t_c(sec)$
Intel Core i5 4278	1	2.7
	2	1.3
Two Six-core AMD Opteron 2345 Processors	1	1.83
	2	1.27
	3	1.03
	4	0.88
	8	0.59
	12	0.49
ARM Cortex-A15 quad core (Odroid XU3 Board)	1	5.20
	2	2.64
	3	1.98
	4	1.5

Table II
CLASSIFICATION TIMES t_C MEASURED AFTER MAPPING THE OPTIMIZED LIDE-C-BASED DNN IMPLEMENTATION ONTO VARIOUS MULTICORE PLATFORM CONFIGURATIONS.

VIII. CONCLUSION

In this paper, we have presented a unified methodology for modeling, mapping, and transforming deep-learning implementations on resource-constrained platforms using dataflow techniques. We have demonstrated this methodology using a design and implementation case study of a deep neural network (DNN) for vehicle classification. Using the lightweight dataflow environment (LIDE), we have applied model-based design methods and using the resulting dataflow representations, we have applied various design optimizations to derive efficient implementations of the targeted vehicle classification system on three different multicore platforms with limited numbers of available cores. Useful directions for future work include extending the developed design methodology to perform joint investigation of trade-offs among DNN complexity, classification accuracy, real-time implementation performance, and resource requirements (cost).

IX. ACKNOWLEDGMENTS

This research was supported in part by the Finnish Funding Agency for Innovation on funding decision 40142/14 (FiDiPro — Advanced Dynamic Stream Processing for Networking and Big Data), and by CSC–IT Center for Science Ltd. for computational resources.

REFERENCES

[1] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, 2nd ed. Springer, 2013.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Proceedings of the Conference on Neural Information Processing Systems*, 2012, pp. 1097–1105.

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

[4] Y. Petetin, C. Laroche, and A. Mayoue, “Deep neural networks for audio scene recognition,” in *Proceedings of the European Signal Processing Conference*, 2015, pp. 125–129.

[5] O. Gencoglu, T. Virtanen, and H. Huttunen, “Recognition of acoustic events using deep neural networks,” in *Proceedings of the European Signal Processing Conference*, 2014, pp. 506–510.

[6] M. Abadi *et al.*, “Tensorflow: Large-scale machine learning on heterogeneous distributed systems,” Google Research, Tech. Rep., November 2015, preliminary White Paper.

[7] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya, “A lightweight dataflow approach for design and implementation of SDR systems,” in *Proceedings of the Wireless Innovation Conference and Product Exposition*, Washington DC, USA, November 2010, pp. 640–645.

[8] H. Huttunen, F. Yancheshmeh, and K. Chen, “Car type recognition with deep neural networks,” *ArXiv e-prints*, February 2016, submitted to IEEE Intelligent Vehicles Symposium 2016.

[9] Y. Jia *et al.*, “Caffe: Convolutional architecture for fast feature embedding,” *arXiv preprint arXiv:1408.5093*, 2014.

[10] C. Shen, W. Plishker, and S. S. Bhattacharyya, “Dataflow-based design and implementation of image processing applications,” in *Multimedia Image and Video Processing*, 2nd ed., L. Guan, Y. He, and S. Kung, Eds. CRC Press, 2012.

[11] N. Sane, H. Kee, G. Seetharaman, and S. S. Bhattacharyya, “Topological patterns for scalable representation and analysis of dataflow graphs,” *Journal of Signal Processing Systems*, vol. 65, no. 2, pp. 229–244, 2011.

[12] J. S. Kin and J. L. Pino, “Multithreaded synchronous data flow simulation,” in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2003.

[13] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly & Associates, Inc., 1996.

[14] S. Kedilaya, W. Plishker, A. Purkovic, B. Johnson, and S. S. Bhattacharyya, “Model-based precision analysis and optimization for digital signal processors,” in *Proceedings of the European Signal Processing Conference*, Barcelona, Spain, August 2011, pp. 506–510.

[15] S. S. Bhattacharyya, W. Plishker, C. Shen, N. Sane, and G. Zaki, “The DSPCAD integrative command line environment: Introduction to DICE version 1.1,” Institute for Advanced Computer Studies, University of Maryland at College Park, Tech. Rep. UMIACS-TR-2011-10, 2011.