

Summer 2014

Nn-X - a hardware accelerator for convolutional neural networks

Vinayak A. Gokhale

Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_theses



Part of the [Computer Engineering Commons](#), and the [Electrical and Electronics Commons](#)

Recommended Citation

Gokhale, Vinayak A., "Nn-X - a hardware accelerator for convolutional neural networks" (2014). *Open Access Theses*. 722.
https://docs.lib.purdue.edu/open_access_theses/722

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Vinayak Gokhale

Entitled

nn-X - A Hardware Accelerator for Convolutional Neural Networks

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

EUGENIO CULURCIELLO, Co-Chair

Chair

ANAND RAGHUNATHAN, Co-Chair

VIJAY RAGHUNATHAN

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): EUGENIO CULURCIELLO, Co-Chair

Approved by: M. R. Melloch

Head of the Graduate Program

06/23/2014

Date

NN-X - A HARDWARE ACCELERATOR FOR CONVOLUTIONAL NEURAL
NETWORKS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Vinayak A. Gokhale

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

August 2014

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

This dissertation and the research it presents would not have been possible without the guidance of Dr. Eugenio Culurciello. Dr. Culurciello has been and continues to be a constant pillar of support. I have learned a great deal from my time working under him and I am extremely grateful to him for giving me this opportunity.

I would like to thank Dr. Anand Raghunathan for giving me advice throughout my Masters degree. Discussions with him have always been inspiring. I would also like to thank Dr. Vijay Raghunathan for his guidance and serving on my advisory committee.

I would like to thank my parents for their constant encouragement. Their guidance and unwavering support has been extremely important throughout my life. I would also like to thank Berin Martini under whose guidance I have learned most of my skills while at Purdue. I am also greatly thankful to the other members at e-Lab - Alfredo Canziani, Aysegul Dundar, Bharadwaj Krishnamurthy and Jonghoon Jin. They have been the source of several thought-provoking discussions. I am also thankful to my friends Shraddha Bodhe, David Miley and Ninad Trifale for making my time at Purdue very enjoyable.

TABLE OF CONTENTS

| | Page |
|--|-----------|
| LIST OF TABLES | v |
| LIST OF FIGURES | vi |
| ABSTRACT | viii |
| 1 INTRODUCTION | 1 |
| 1.1 Convolutional Neural Networks | 2 |
| 1.1.1 The Convolution Operator | 2 |
| 1.1.2 The Non-linear Operator | 3 |
| 1.1.3 The Pooling Operator | 4 |
| 1.1.4 Flow of Information Through a ConvNet | 4 |
| 1.2 Outline | 5 |
| 2 LITERATURE REVIEW | 7 |
| 3 COPROCESSOR ARCHITECTURE | 10 |
| 3.1 The Host Processor | 11 |
| 3.2 Collections | 12 |
| 3.2.1 Convolution Engine | 13 |
| 3.2.2 Non-linear Operator | 14 |
| 3.2.3 Pooling module | 14 |
| 3.3 Memory Router | 15 |
| 3.4 Configuration Bus | 16 |
| 4 SYSTEM OVERVIEW | 18 |
| 4.1 Overview of a DMA Transaction | 18 |
| 4.2 Flow of Data Within the Coprocessor | 21 |
| 5 PERFORMANCE COMPARISON AND ANALYSIS | 25 |
| 5.1 Performance per Resource | 26 |

| | Page |
|--|------|
| 5.2 Performance Per Unit Power Consumed | 27 |
| 5.3 Raw Performance | 29 |
| 6 DISCUSSION | 34 |
| 6.1 Lack of Control Flow | 34 |
| 6.2 Efficiency of Memory Accesses | 35 |
| 6.2.1 N = 1 case | 36 |
| 6.2.2 N > 1 case | 37 |
| 7 FUTURE WORK AND CONCLUSION | 39 |
| 7.1 Future Work | 39 |
| 7.2 Conclusion | 40 |
| LIST OF REFERENCES | 42 |

LIST OF TABLES

| Table | Page |
|---|------|
| 5.1 This table describes nn-X's hardware specifications | 25 |

LIST OF FIGURES

| Figure | Page |
|--|------|
| 1.1 Architecture of a typical convolutional neural network for object recognition: a convolutional feature extractor followed by a classifier (like a multi-layer perceptron) for generic multi-class object recognition. Once trained, the network can parse arbitrarily large input images, generating a classification map as the output. | 2 |
| 3.1 A block diagram of the nn-X system. nn-X is composed of a coprocessor, a host processor and external memory. The coprocessor has three main components: processing elements called collections, a system bus called the memory router and a configuration bus to control flow of data. Collections perform the most typical operations in ConvNets: convolutions, pooling and non-linearity. | 11 |
| 3.2 A collection comprises a router and three operators. The router has “all-to-all” connections forming a crossbar switch. The configuration bus forms the select line for the mux-demux combination. | 12 |
| 3.3 The memory router makes two connections with each collection and two with the HP ports. Each bus in the figure is bi-directional. | 15 |
| 4.1 A pictorial representation of the state of the memory before initiating a DMA transaction. The system memory is a buffer that holds a JPEG image and the image array has 32-bit elements. Each channel is stored separately in DMA buffers and has 8-bit elements. | 19 |
| 4.2 Components involved in performing a DMA transaction. Data is stored in memory by the host processor. From here, a DMA transaction is initiated by the host. The DMA engines, which are soft IP and are implemented in the programmable logic, receive data from memory and store it in a buffer. From here, data is transferred to the coprocessor which is also in the programmable logic. | 20 |
| 4.3 Each bus has three signals — data, valid and ready. When valid is asserted, the data on the bus can be read and processed. Valid can only be asserted when the ready is high. Ready is controlled by the destination; it is high when the destination — DMA engine or coprocessor — is ready to receive data. The clock is the AXI clock and is the primary clock for all peripherals on the AMBA bus. | 21 |
| 4.4 Packing of data within the coprocessor. | 23 |

| Figure | Page |
|---|------|
| 5.1 Performance of the nn-X coprocessor with increasing collections. The input was a 500×500 image to a 8 collections, each featuring 10×10 disparate convolution kernels. | 27 |
| 5.2 Performance per watt of different platforms. Most desktop CPUs and GPUs gave under 3 G-ops/s-W while mobile processors performed slightly better. nn-X (red) implemented in programmable logic was able to deliver more than 25 G-ops/s-W. | 28 |
| 5.3 Single neural network layer with 4 input planes of 500×500 , 18 output planes and 3.6 billion operations per frame. nn-X computed one frame in 6.2 ms and was 271 times faster than the embedded processors. | 29 |
| 5.4 Performance comparison across embedded devices when running the filter-bank test. | 30 |
| 5.5 A face detector application with 552 M-ops / frame. nn-X was able to process a 500×350 video at 42 frames a second and was 115 times faster than the embedded processors. The image on the left is a multi-scale pyramid to provide scale-invariance to the input. | 31 |
| 5.6 Comparison of performance across embedded devices. nn-X performs more than 4 times better than the next fastest processor on the face detector. | 32 |
| 5.7 Street scene-parser requiring 350 M-ops / frame. This application processes a 510×288 input in 4.5 ms and produces an 8-class label for each frame. On this application, nn-X is 112 times faster than the Zynq processors. | 33 |
| 5.8 Comparison of performance across embedded devices. nn-X performs at least 3.5 times better than the embedded processors. | 33 |
| 6.1 An example of 1-to- M and N -to- M convolution. These are the two most commonly encountered cases in ConvNets. A 2D diagram is used to visualize the relation between the convolution layers for simplicity. Circles indicate convolutional planes. M and N are 4 and 2 respectively. | 36 |
| 6.2 An example routing of a network producing one output from three inputs. The input to each router comes from the memory router. The output of the left-most operator block is a convolution (orange). It gets sent to the center collection where it is combined with the convolution of the blue input stream to produce the purple stream as another intermediate. This stream gets sent to the collection on the right to get combined with the convolution of the yellow input to produce the red result as the final output which is then sent to memory. The connections between neighbors facilitate rapid data transfer between collections. | 37 |

ABSTRACT

Gokhale, Vinayak A. M.S.E.C.E, Purdue University, August 2014. nn-X - A Hardware Accelerator for Convolutional Neural Networks. Major Professor: Eugenio Culurciello.

Convolutional neural networks (ConvNets) are hierarchical models of the mammalian visual cortex. These models have been increasingly used in computer vision to perform object recognition and full scene understanding. ConvNets consist of multiple layers that contain groups of artificial neurons, which are mathematical approximations of biological neurons. A ConvNet can consist of millions of neurons and require billions of computations to produce one output.

Currently, giant server farms are used to process information in real time. These supercomputers require a large amount of power and a constant link to the end-user. Low powered embedded systems are not able to run convolutional neural networks in real time. Thus, using these systems on mobile platforms or on platforms where a connection to an off-site server is not guaranteed, is unfeasible.

In this work we present nn-X — a scalable hardware architecture capable of processing ConvNets in real time. We evaluate the performance and power consumption of the aforementioned architecture and compare it with systems typically used to process convolutional neural networks. Our system is prototyped on the Xilinx Zynq XC7Z045 device. On this device, we are able to achieve a peak performance of 227 GOPs/s, a measured performance of up to 200 GOPs/s while consuming less than 3 W of power. This translates to a performance per power improvement of up to 10 times that of conventional embedded systems and up to 25 times that of performance systems like desktops and GPUs.

1. INTRODUCTION

Embedded vision systems find applications in many areas - autonomous robots, security and surveillance systems, UAVs and more recently, mobile phones and automobiles. These applications require algorithms that are accurate and can be processed in real time. Furthermore, their embedded nature requires low power consumption. Many recent algorithms have been proposed for performing object recognition and visual understanding. SIFT feature extractor and the SURF algorithm that it inspired [1, 2], models of the human visual cortex (HMAX) [3] and convolutional neural networks (ConvNets) [4–8] are some models that perform feature extraction for object recognition, detection and scene understanding.

Of these, convolutional neural networks achieving state-of-the-art perception [4, 9] and being applied to artificial intelligence [10] have recently been presented. These models' success has prompted their use by companies like Google and Baidu [11] for visual search engines. ConvNets comprise hundreds of convolutional filters across multiple layers and require billions of computations to process one frame. This requires the use of high throughput, power intensive processors and GPUs to be able to process these networks in real time. For mobile applications like on smartphones, data is sent to off-site server banks and the results are then sent back over a network link. This requires a constant, reliable connection to the off-site server, a fact that limits the abilities of these mobile systems. High-risk tasks like self-driving vehicles or security systems are not be able to use the capabilities provided by ConvNets due to lack of custom hardware designed specifically to process these networks in real time on such systems.

In this thesis we present *nn-X*, a scalable custom hardware architecture that is capable of processing convolutional neural networks in real time. *nn-X* is a low-powered mobile system for accelerating convolutional neural networks. The *nn-X* system comprises a host processor, a coprocessor and memory. The *nn-X* coprocessor efficiently implements pipelined operators and exploits a large amount of parallelism to deliver very high perfor-

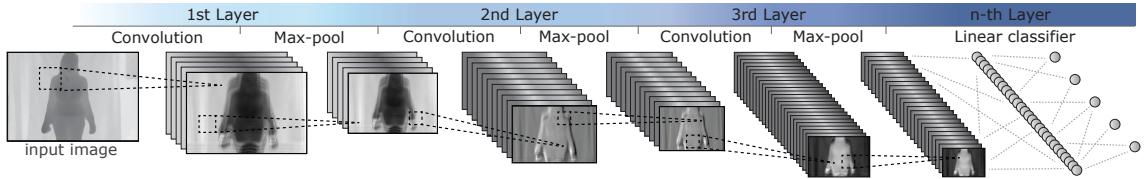


Fig. 1.1.: Architecture of a typical convolutional neural network for object recognition: a convolutional feature extractor followed by a classifier (like a multi-layer perceptron) for generic multi-class object recognition. Once trained, the network can parse arbitrarily large input images, generating a classification map as the output.

mance per unit power consumed. The prototyping platform used in this work consumes 8 W of power for the entire platform and only 3 W for the nn-X system and memory.

1.1 Convolutional Neural Networks

A ConvNet comprises several convolution layers. These are followed by a classifier that classifies the outputs of the convolutional layers as one of the multiple objects it is trained on. A typical ConvNet is shown in Figure 1.1.

Each convolution layer is followed by a pooling operation and a non-linearity. In this work, we consider a *layer* of a ConvNet to comprise all three of the above mentioned operators while a *convolution layer* consists of only the convolution operator. Inputs of a layer are typically images (or frames from a video) but can be any kind of locally correlated data, like audio signals. The outputs of one layer act as inputs to the next. The inputs (and by extension, the outputs) are called *feature maps*.

1.1.1 The Convolution Operator

Mathematically, convolution is an operation performed on two functions that outputs a third function which is essentially a modified version of one of the two input functions. This operation is given in Equation 1.1 below

$$y[m, n] = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} x[m+i, n+j] \cdot w[i, j] \quad (1.1)$$

where $y[m, n]$ is one data word in the output, $x[m, n]$ is an input data word and $w[m, n]$ are values of the filter kernels.

In ConvNets, convolution acts to extract a particular type of feature in the input [12]. The input consists of data containing the features to be extracted. Inputs can be audio, video or raw data and are typically several thousand samples. In this work, we primarily focus on image inputs.

The convolution kernel acts as the second function. Each kernel is trained to recognize a particular feature by filtering out the rest of the image. The kernel can be any type of filter. For example, a Gabor kernel acts as a band pass filter and will recognize edges that occur at angles similar to its own orientation. A network trained to recognize faces can consist of hundreds of such filters oriented at different angles. These can then extract the various features of the human face.

1.1.2 The Non-linear Operator

The non-linearity is used as an activation function to convert the input space into linearly separable output spaces. This operator acts to model the rate of firing of the action potential in a biological neuron. Piece-wise linear activation functions like the Heaviside step function can be used for this purpose but then linearly inseparable input spaces cannot be processed by the network [13].

For this reason, non-linear sigmoid functions - specifically, the hyperbolic tangent - became popular with ConvNets to serve as activation functions. More recently, the rectifier function has become popular [4].

1.1.3 The Pooling Operator

Pooling primarily serves the purpose of providing translational invariance to an input. By pooling over a two dimensional region, we gather information from that region into one output. If a pixel within that region translates over the pooling region, the output will be identical.

A second purpose of the pooling operator is to reduce the size of the feature maps for subsequent layers. The number of feature maps increases as we go deeper into the hidden layers. Smaller feature maps reduce the computational complexity.

1.1.4 Flow of Information Through a ConvNet

The ConvNet shown in Figure 1.1 takes a greyscale input. In this case, the first layer would require N convolutions to produce N outputs, where $N > 1$. Here, the input image is convolved with each convolution kernel exactly once. The N feature maps produced as a result are then sent in as inputs to the next layer.

Layer 2 can be fully connected or partially connected. A fully connected layer implies that information from all of its inputs is used to produce every output. In a partially connected layer, a particular output might not contain information from a particular input but every input is taken into account in some output.

In this example, we will consider the fully connected case with layer 2 having M outputs. Each output is the result of combining N intermediate results. This requires that there be $N \times M$ kernels in layer 2 — N sets of M kernels. The output produced by the convolution of one feature map with one kernel is called an *intermediate result*. The first input feature map is convolved with the first kernel of each set to produce N intermediate results. These N intermediates are combined to produce one output feature map. The combination is done by adding the i -th word of each intermediate result to produce the i -th word of the output. Similarly, the other $N - 1$ input feature maps are convolved with their respective kernels to produce the other $M - 1$ output feature maps.

The kernels can be of any size but are typically between 7×7 and 11×11 for the first layer. The subsequent layers have smaller kernels but this is not necessary. It is typically done to reduce both training and processing times.

For a layer with 9×9 kernels, 3 input feature maps, 96 output feature maps and an input image of 224×224 , 2.17 *billion* operations are required to produce the 96 outputs. This is the first layer of the recently popular network described by Krizhevsky in [4]. Subsequent layers can require even more computations. Finally, this produces an output classification map for a single image. Most video runs at 24 frames per second although over 10 is considered fast enough to be real-time [14]. Depending on the application, faster processing times could be required. As is evident from the numbers above, several billion computations are required per second to process video when using large networks.

1.2 Outline

This section describes the organization of this document. In the next chapter, we do a review of current and past architectures that target the hardware acceleration of artificial vision, with some specific architectures targeting ConvNets. In Chapter 3, we describe the coprocessor architecture. Chapter 3 focuses on the design of the coprocessor and the rationale behind its structure. We introduce the mathematical operators implemented in the coprocessor and describe the life cycle of data within the coprocessor when running a typical feed-forward ConvNet. In Chapter 4, we describe the rest of the system that builds around the coprocessor. We also describe the life cycle of data from the time of its inception to its consumption by the coprocessor and the life cycle of the resulting output. This chapter completes the description of the entire nn-X system. Chapter 5 demonstrates the throughput and performance per unit power consumed of the nn-X system. We also compare these metrics to both embedded and high performance systems commonly used to run feed-forward networks. In Chapter 6, we investigate the performance gains of the coprocessor. We also theorize the reasons for sub-optimal performance of general purpose

processors. Finally, Chapter 7 concludes this thesis by summarizing the nn-X system, the results and briefly describing the future directions for this project.

2. LITERATURE REVIEW

Convolutional neural networks and, really, deep networks in general have been proposed for over two decades. However, only recently have they been put to use in real-world applications. This is primarily because of the massive amount of processing power required to train and process these networks. Such computational throughput was not available on general purpose architectures until very recently. As such, smaller networks were used but their applications were limited. Networks like those used by Google for visual searches are huge and require several billion operations per frame. Their capabilities are made available to the end user by processing data off-site on servers and sending results back over the network.

To that end, several custom architectures have been proposed for embedded processing of neural networks. This chapter details some of these architectures.

We identify three major design points that virtually all existing architectures have taken into account — scalability, programmability and communication or I/O.

Scalability is important because all platforms need not run the same ConvNet architectures. Some platforms, like security systems might not need to run a network as large as one that is needed for a self-driving car. Nor would it need to be processed as fast. Such a system would need a smaller number of processing elements that consume lower power than a system that needs to process larger networks or process them faster.

Programmability is necessary for the same reason a general purpose processor is programmable. It allows an end user to write software (in C, for example) without knowing the details of the hardware architecture. Furthermore, while the convolution operator has remained static through generations of ConvNets, the activation function and the pooling operator have changed through the years. ConvNets usually featured sigmoid functions as activation functions. More recently, Krizhevsky et. al. have described the threshold operator to perform favorably [4].

Communication is required by an accelerator because it needs to send the results it has produced to the appropriate location to take action. For example, if the processor detects an obstacle in the path of a self-driving car, it needs to send a signal to the engine control unit to slow the vehicle down. Alternatively, the architecture could be designed as a coprocessor which does not interpret the results but simply sends them out to a host processor which is responsible for interpreting results and taking appropriate actions. This latter topology is becoming more popular due to its nature of being modular and easily integrable with a variety of platforms.

An early implementation of an architecture targeted towards neural networks was described by Cloutier in [15]. While slow by today's standards, the VIP implementation described by Cloutier was very efficient and several times faster than the processors of the day. Cloutier describes an architecture with a grid of *processing elements* (PE) that perform the convolution operations. Furthermore, this grid is part of a larger grid of four FPGAs which are part of a 2×2 two dimensional mesh. A PCI bus is used to communicate with a host processor. Almost all architecture targeted towards ConvNets that have followed the one described by Cloutier use the concept of *single instruction, multiple data* (SIMD). SIMD allows us to exploit the inherent parallelism of the convolution operator that lies at the heart of all ConvNets.

Kapasi describes an architecture that employs a separate instruction set [16]. An external host is used to send compiled instructions to the processor and results are sent out. On-chip memory is used as a staging area to both, store results and access inputs. Efficient use of such an architecture dictates for the use of large amounts of SRAM or large amounts of data transfer between processor and off-chip memory. The former is inefficient in area while the latter is inefficient in throughput. SRAM costs have not seen a significant decrease with time (as compared to DRAM) [17] and DRAM accesses are slow.

Research by Farabet has produced a coprocessor that delivers high performance at low power consumption. A prototype system is described in [14, 18]. An ASIC implementation of this architecture was described in [19]. This architecture is scalable and programmable. However, it suffers from slow host-coprocessor data transfer. An ethernet link is used to

interface with the host which adds overhead to off-board data transfer. Finally, an all-to-all connection grid is used between its operators and while this adds flexibility, it does not necessarily provide better throughput if these connections are seldom utilized. A large number of such connections are inefficient in an ASIC implementation and reduce maximum operating frequency.

3. COPROCESSOR ARCHITECTURE

As seen in the previous chapter, a variety of architectures have been used when designing custom processors for ConvNets. Also introduced were the three design points that were important when designing a custom coprocessor — scalability, programmability and communication with the host.

The nn-X system described here takes all three points into consideration. The coprocessor is intended to function alongside the host on the same board, or even the same chip, the latter making it a truly heterogeneous system. A block diagram of the nn-X system is shown in Figure 3.1. In this implementation, the coprocessor is on-chip.

The nn-X system has three main components: *a host processor*, *a coprocessor* and *external memory*. The coprocessor comprises an array of processing elements called *collections*, *a memory router* and *a configuration bus*. The collections are a group of functional units required to perform the most typical operations in ConvNets — convolutions, pooling and non-linear functions. Each collection also contains a local router for managing data flow within the collections and interfacing with the memory router.

The coprocessor is a completely modular system. It can function alongside any host processor with any instruction set and any architecture. It is not bandwidth bottlenecked when interfacing with the host as communication is primarily through the shared memory (and as such, bottlenecked by the host's memory controller itself).

The coprocessor uses a 16-bit data bus that follows the Q8.8 number format. This format has been shown to provide virtually identical results to neural networks implemented in 32-bit floating point [20–22].

The nn-X coprocessor is implemented on the embedded programmable logic and interfaces with the host via the AXI bus. Input data in the form of 2D planes is streamed into the nn-X coprocessor, one data word per clock cycle. Data is organized as an array,

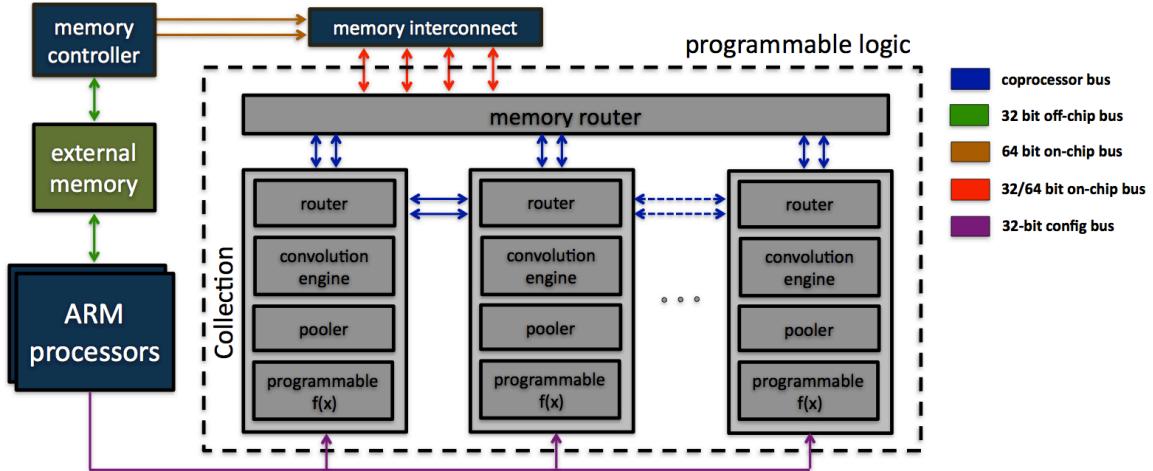


Fig. 3.1.: A block diagram of the nn-X system. nn-X is composed of a coprocessor, a host processor and external memory. The coprocessor has three main components: processing elements called collections, a system bus called the memory router and a configuration bus to control flow of data. Collections perform the most typical operations in ConvNets: convolutions, pooling and non-linearity.

with data words streamed in one row at a time. This chapter describes in detail the primary components of the nn-X system.

3.1 The Host Processor

Two ARM Cortex-A9 CPUs function as the host processor for the nn-X implementation described here. The host runs the Linux operating system. The processor is responsible for parsing a network, compiling it into configuration instructions for the coprocessor and processing operations that are not implemented on the coprocessor. The compiled instructions are stored in memory. These instructions are created at compile time and do not change throughout the program.

The host also controls transfer of input and configuration data to the coprocessor. This is done over the AXI4-Lite bus and is described in Section 3.4.

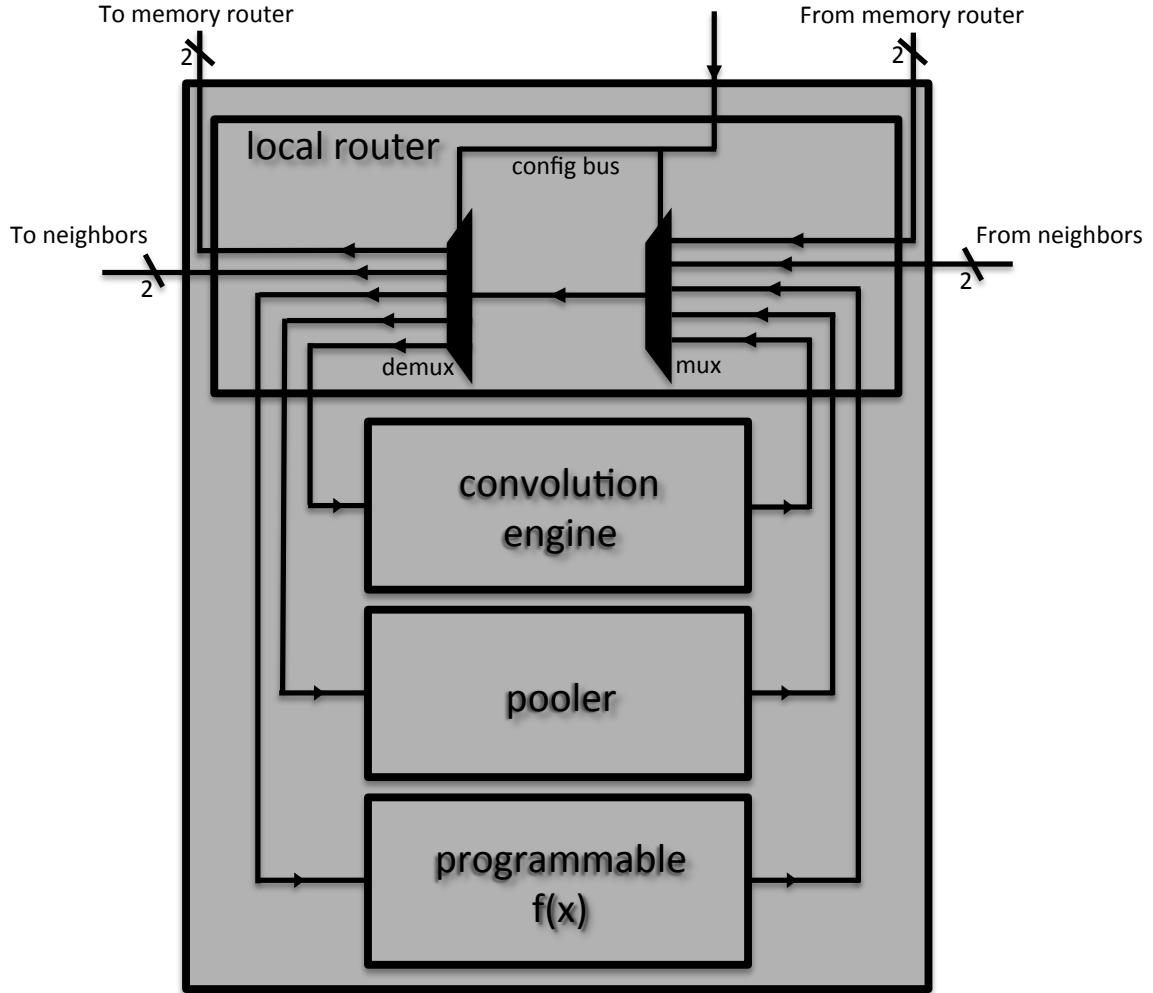


Fig. 3.2.: A collection comprises a router and three operators. The router has “all-to-all” connections forming a crossbar switch. The configuration bus forms the select line for the mux-demux combination.

3.2 Collections

Each collection comprises of: one *convolution engine*, one *pooling module* and one *non-linear operator*. Figure 3.2 shows a collection in detail.

As seen in the figure, each collection has a local router to direct data to the desired operator, to neighboring collections or back to the memory router. All connections can be simultaneously enabled. All operators are pipelined resulting in one output word produced

every clock cycle, notwithstanding an initial setup delay. The following subsections define the flow of data in a generic ConvNet of the type described in [4].

3.2.1 Convolution Engine

Convolution is the most typical operation of the eponymous convolutional neural networks. Convolution is inherently parallel and can be accelerated on data parallel architectures. The operation was introduced in Chapter 1, Section 1.1 and is given below:

$$y[m, n] = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} x[m + i, n + j] \cdot w[i, j] \quad (3.1)$$

In Equation 3.1, $y[m, n]$ is one data word in the output, $x[m, n]$ is an input data word and $w[m, n]$ are the weights of the filter kernels.

When a convolution needs to be performed, the weights are first streamed in. These weights are cached for the entire duration of the operation. The nn-X implementation described here supports a reprogrammable kernel size of up to 10×10 .

The convolution engine is implemented as fully pipelined logic and uses memory to cache incoming data. This cache is needed for pipelined implementation of the convolution operation [14]. For a row of width W and a $k \times k$ convolution filter, the size of this cache is $W \times k \times 2$ bytes. The factor of 2 comes from the fact that each data word is two bytes wide. After a delay that is equal to the depth of this cache, outputs are available every clock cycle. This allows the system to have the max allowable data width as a design parameter.

Output data can then be routed to other operators in the same collection to perform cascaded pipelined sequences of operations. It can also be sent to a neighboring collection to be combined with the output of a convolution performed there.

The convolution engine can also perform pooling operations. The kernel can be used to implement a smooth pooling function (for example, Gaussian) or perform a running average of pixels or data words (with a uniform kernel).

3.2.2 Non-linear Operator

The non-linear operator computes a piece-wise linear approximation of any arbitrary non-linear function. The non-linear operation is described in equation (3.2).

$$y(x) = a_m x + b_m \quad \text{for } x \in [l_m, l_{m+1}) \quad (3.2)$$

where l_m is the lower bound of the m -th segment and a_m and b_m are its slope and y -intercept.

The non-linear operator can be programmed to approximate the typical non-linear functions used in ConvNets like a logistic sigmoid, absolute value and the recently popularized rectifier unit, i.e. $\max(x, 0)$ [4]. The number of linear segments used is a design parameter. This affects the precision of smooth, non-linear function approximations.

This functional unit is configured with the function to implement at the beginning of a computation. The function is cached until either the unit is reconfigured or reset. To configure this unit, multiple configuration packets are sent in. The number of configuration packets is equal to the number of linear segments used. Each packet consists of the lower bound of the segment (which dictates what range of inputs map to it), its y -intercept and its slope.

3.2.3 Pooling module

In ConvNets, pooling of data is necessary to give translational invariance to the input. Pooling also results in reduction of the size of the output feature maps produced by each layer. nn-X includes a special max-pooling module that takes into consideration a $p \times p$ 2D region of the input and outputs the maximum value as the result.

nn-X's max-pooling module requires one digital comparator and a small amount of memory. The input data is streamed into the module one row at a time and the max operation is computed by first computing the maximum of each $\lfloor \frac{W}{p} \rfloor$ group of data words in the row, with W being the width of each row. As data words are streamed in, each group of p

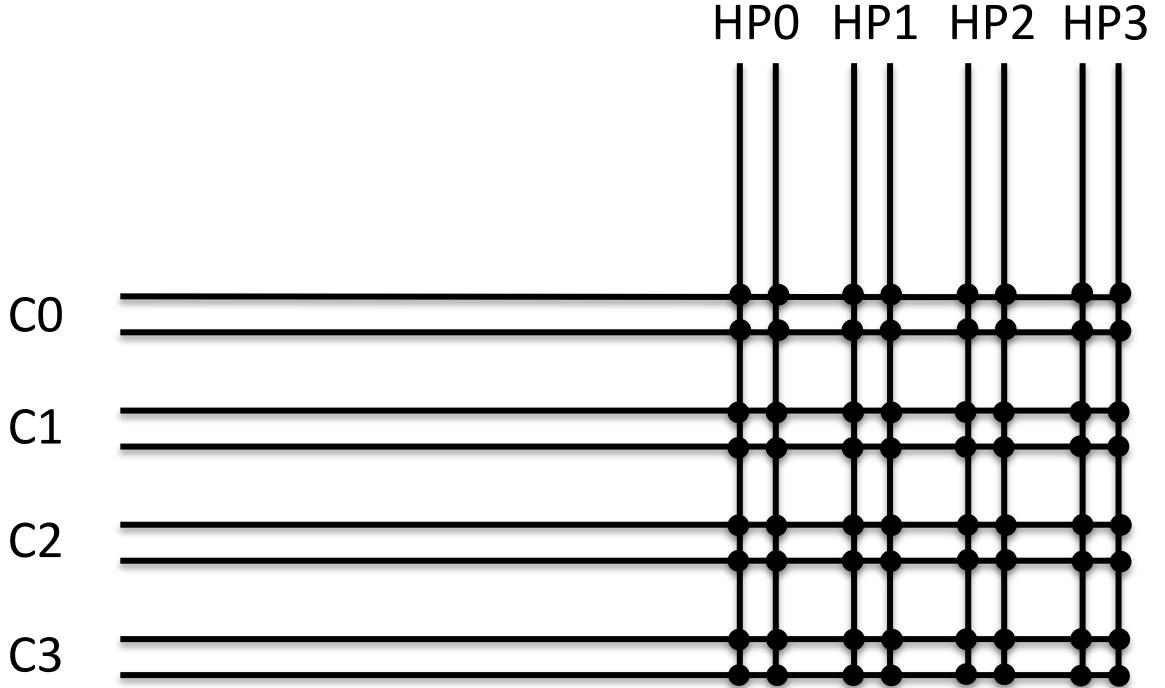


Fig. 3.3.: The memory router makes two connections with each collection and two with the HP ports. Each bus in the figure is bi-directional.

is compared to the previous max value stored in memory. This requires storing $\lfloor \frac{W}{p} \rfloor$ values into the local cache, as the first output cannot be computed until the first p data words of the p -th row are streamed in. After operating on p rows, the final output can be computed and output values start to stream out of the module. The value of p can be configured at run-time and changed as needed.

The advantage of this implementation is that it requires a very small amount of memory to compute the maximum over a 2D region. In fact, the total memory required is equal to W , the maximum width of the input image.

3.3 Memory Router

The memory router interfaces the collections with memory. Its purpose is to route independent data streams and feed data to the collections. The router is implemented as

a crossbar switch, allowing nn-X access to multiple streams at once and performing full-duplex data transactions, as shown in Figure 3.3.

There is a small amount of logic between the router's outputs and memory. This is described in detail in the next chapter. The router interfaces with the Zynq's AXI memory interconnect, which allows for up to four DMA channels with an aggregate bandwidth up to 3.8 GB/s.

DMA transactions to and from memory are initiated by a custom Linux device driver. This driver enables nn-X to initiate up to four simultaneous bidirectional transactions at a given time. nn-X uses register polling to determine when a transaction is complete. The process of initiating a DMA transaction and the life cycle of a data stream is described in detail in the next chapter.

3.4 Configuration Bus

The AMBA 4 bus has an AXI 4-Lite data transfer specification which is a low throughput, memory-mapped bus. 32 registers are available to send and receive data from the coprocessor. It takes exactly 16 cycles once a transaction is initiated to transfer data to the coprocessor. Each transaction can transfer a single 32-bit data word.

A custom Linux character driver is used to transfer configuration data to the coprocessor. The configuration data can be one of the following:

- Resets - The resets are so organized that individual areas can be reset separately as, of course, can the entire coprocessor. For example, if only one collection needs to be reset, or only the memory router, these can separately be reset without having to restart the rest of the coprocessor. This allows for addition of caches, among other things, which would not be able to hold data if only a global reset was available.
- Configuration of data paths - Not all data needs to follow the exact same path. For example, intermediate results need to be stored in memory until all intermediates have accumulated. The output is then sent to the max-pool and then the non-linearity functional units. The memory router also needs to be configured to receive streams

from memory and send them to the collections. Finally, the collection router needs to be configured to receive data from the memory router and direct it to a functional unit or the neighboring collection. Alternatively, on the return path, it can be configured to receive data from a functional unit and send to the neighboring collection or back to the memory router.

- Functional unit configuration - Each functional unit needs to be configured before it can process a stream. The convolver needs to be configured with the size of the kernel and the image, the max-pool unit needs to be configured with the pooling area and the non-linear unit needs to be configured with the non-linear function it needs to implement.
- Convolution kernels - The weights for a convolution are transferred over the configuration bus. Once the weights are cached, the convolver is ready to receive an image.

4. SYSTEM OVERVIEW

This chapter describes the life cycle of an image in the nn-X system. It details the flow of data, starting from its storage in memory to being processed by the coprocessor and ending with the generation of a classification map and its associated results.

ConvNets process the red, green and blue channels of an image separately. Most images from cameras or those stored on disk are compressed using the JPEG standard. Decompressed JPEG images have 24 bits per pixel, 8 bits each for the red, green and blue (RGB) channels. This allows us to decompress the image into red, green and blue image streams and store each channel as a separate array in memory. Each channel is then sent in to the accelerator independently. This is shown in Figure 4.1. The figure shows a pictorial representation of the state the memory is in before a DMA transaction. The JPEG frame is stored in system memory. It is then separated into its three channels and stored in the DMA buffers. While this involves a memory copy, this latency can be hidden within the processing of the previous frame by using a different thread to perform this memory copy.

4.1 Overview of a DMA Transaction

Image data can either be stored on disk or received via I/O (like from a USB camera). From here, it is stored in the system memory as shown in Figure 4.1. For the purposes of DMA, *a physically contiguous, non-cacheable memory buffer* is allocated in memory. Linux has provisions for requesting such a buffer by specifying such a requirement in the *device tree*.

While this memory is set aside by the operating system, its use by software must be explicitly stated. The compiler requests space in this buffer at compile time. Data is then copied from its current location into the DMA buffer. Linux provides functions that give the user a physical handle to the virtual memory location that the data is stored at. Since this

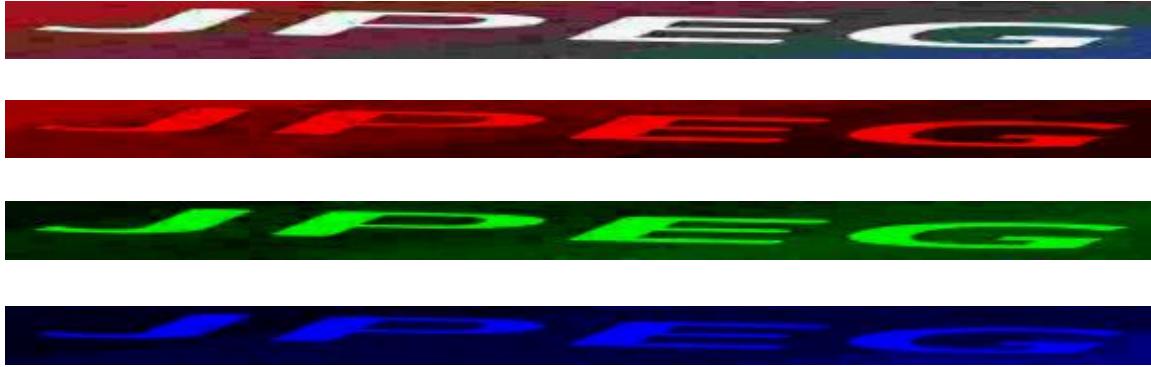


Fig. 4.1.: A pictorial representation of the state of the memory before initiating a DMA transaction. The system memory is a buffer that holds a JPEG image and the image array has 32-bit elements. Each channel is stored separately in DMA buffers and has 8-bit elements.

buffer is guaranteed to be contiguous in physical memory, the DMA engine only requires the physical address of the first element of the data array and the length of data to be transferred.

Xilinx provides soft DMA IP (called DMA engines) to perform transactions over the AXI bus from memory to the programmable logic. Figure 4.2 shows the components involved in a DMA transaction. Four high performance (HP) ports are available to transfer data to the programmable logic. The DMA engines attach themselves to these HP ports. A HP port is essentially a high throughput bi-directional link to memory. It can be programmed to be 32-bit or 64-bit wide. In this implementation, we use 32-bit ports because we do not need the added bandwidth provided by the wider links.

The engines include a buffer to store data until it is required by the coprocessor. A DMA transaction initiated by the software causes the DMA engine to send data to the programmable logic. The software needs to specify the pointer to the first item in the array to be transferred, the length of the array to be transferred and the DMA engine (and by extension, the HP port) that is responsible for the transfer. Data is transferred by the DMA engines from memory in bursts. The DMA engines can perform a burst of 256 data words. Each burst has a setup time associated with it, after which the 256 data words are sent to the DMA engine's buffer, one word per clock cycle.

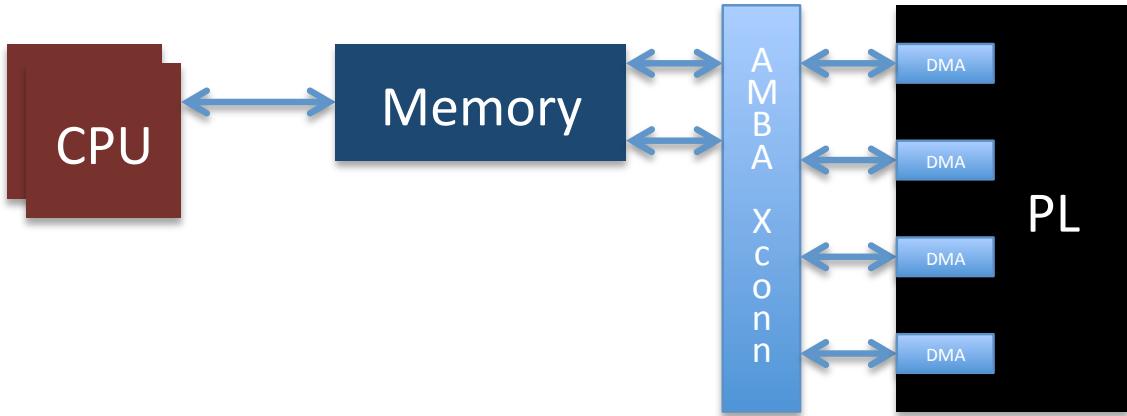


Fig. 4.2.: Components involved in performing a DMA transaction. Data is stored in memory by the host processor. From here, a DMA transaction is initiated by the host. The DMA engines, which are soft IP and are implemented in the programmable logic, receive data from memory and store it in a buffer. From here, data is transferred to the coprocessor which is also in the programmable logic.

The coprocessor has four buses for receiving data from and sending data to the DMA engines. Figure 4.3 shows the signals comprising a bus. Each bus has three signals — *data*, *valid* and *ready*. While the valid signal has the same direction as the data and is a single bit indicating the presence of valid data on the bus, the ready signal has the opposite direction. The ready is a signal coming from the receiver indicating the ability of the receiver to process incoming data. The clock is passed to all peripherals that are part of the AMBA bus.

Processed data is sent to a DMA engine, usually the one the input data came from. Polling registers, one per HP port, are used to determine when the processed data has been written back to memory. From there, another transaction can be initiated.

Alternatively, the data can also be accessed by the host. If the data stored is the final result, this result can be processed by the host. The host can then take appropriate action based on the result.

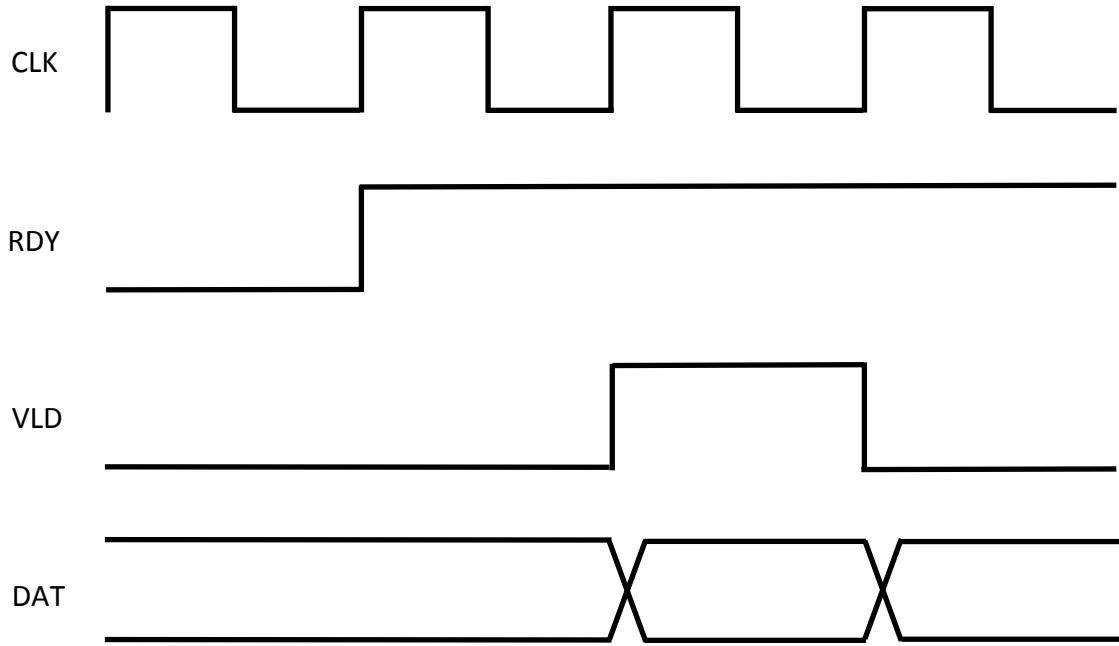


Fig. 4.3.: Each bus has three signals — data, valid and ready. When valid is asserted, the data on the bus can be read and processed. Valid can only be asserted when the ready is high. Ready is controlled by the destination; it is high when the destination — DMA engine or coprocessor — is ready to receive data. The clock is the AXI clock and is the primary clock for all peripherals on the AMBA bus.

4.2 Flow of Data Within the Coprocessor

Inside the coprocessor, data is sampled on the rising edge of the clock, when valid is asserted. While the input and output buses on the HP ports have data widths of 32-bits, the coprocessor uses a 16-bit data bus. This provides for an attractive opportunity to improve throughput and performance. Two 16-bit words can be packed into one 32-bit word. This effectively reduces the number of data words transferred by the DMA engines by a factor of 2. Since the DMA engines have an overhead associated with each transaction and this overhead is directly proportional to the length of the transaction, reducing the length of the DMA transaction while keeping throughput the same will result in a performance improvement. Since two data words are produced every clock cycle, they need to be consumed every clock cycle. This is not possible unless the operators are running at a faster clock frequency (up to a maximum of twice the clock frequency).

However, if the two data words are consumed by two *different* collections, the entire accelerator can run at the same frequency while consuming both data words in the same clock cycle. This is not possible when the RGB channels are input at the beginning but the first three transfers form a small part of the entire process. When data is sent back to memory, one 16-bit word from one collection and one from another collection can be concatenated together. This outgoing data stream contains information from two collections. Since eight collections can fit in the current implementation on the ZC706 development board and there are four HP ports available, concatenation provides a convenient way of utilizing all available resources.

For the purposes of concatenating data, a *packer module* is attached to each HP port. Each packer has a buffer associated with it that is 64 entries deep. The purpose of this buffer is to provide a continuous flow of data in the event that any of the functional units downstream deassert their ready signal for a few clock cycles. Without this buffer, the DMA engine would need to be stalled each time the ready is deasserted, even for one clock cycle. Restarting a transaction from the DMA engines requires a setup time which would be significant in the event that several such one clock cycle delays are requested by the functional units. However, a buffer would hold the data in such cases and these short pauses would be invisible to the DMA engines.

The packer has multiple “modes” for different types of data packing:

1. 8×4 - This mode is used for deserializing four 8-bit words packed into one incoming packet. This is used when passing a red, green or blue channel the first time since each pixel in these channels is 8-bits.
2. 16×2 - This mode is used for deserializing two 16-bit words. This mode is unused for the implementation described here but is included for potential future improvements to the coprocessor including running parts of the coprocessor at a faster clock, which would add the ability to consume two data words in one clock cycle.

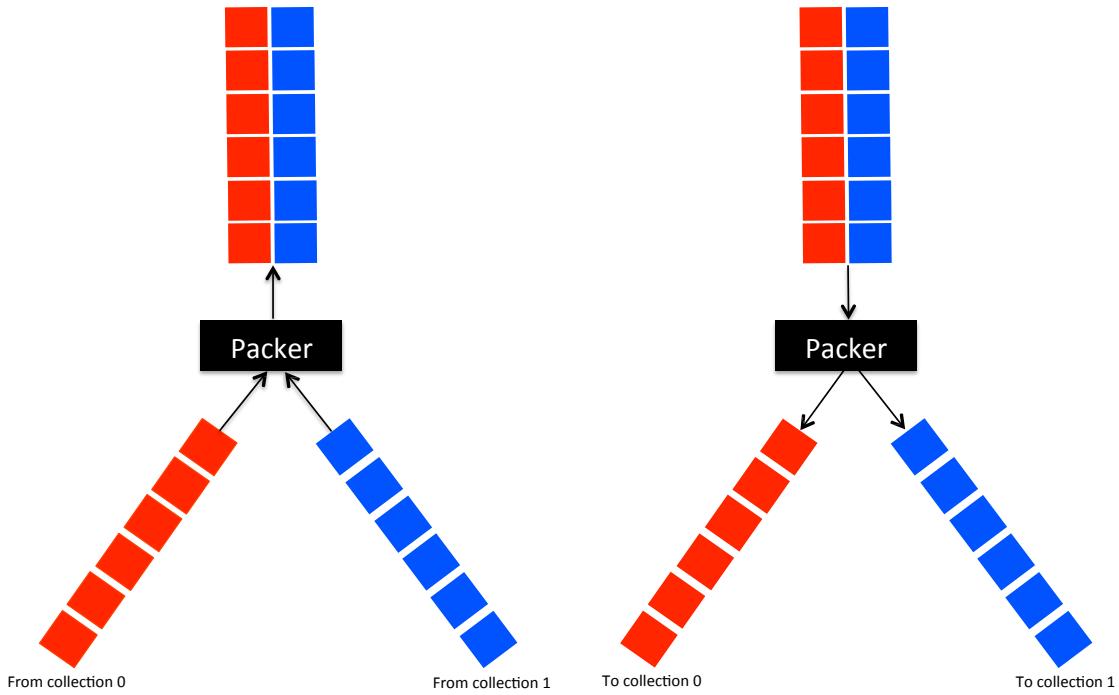


Fig. 4.4.: Packing of data within the coprocessor.

3. Concatenated 16×2 - This mode is used for deserializing two different streams packed into one input stream. This is used for streams except the initial input and the final output.
4. Fixed-to-float conversion - This mode is used when the final output is produced by the coprocessor. This data needs to be read by the host CPU. Since the coprocessor uses fixed point representation, it is much quicker to convert the output stream to the IEEE 754 floating point representation in hardware than have the CPU do the conversion for each pixel. The conversion in hardware takes only one clock cycle and has an initial latency of four clock cycles.

The implementation described here primarily uses modes 1, 3 and 4 described above, at different stages of processing a frame. On the very first pass, data is packed as four 8-bit words as explained in Section 4.1. However, all this data goes to the same collection (since each stream is not a concatenation of two different data streams). When it is processed

and sent back, eight collections are feeding data back at the same time. Data from two collections is concatenated into one 16-bit word and sent out as one data packet. When the streams are read out from memory, they are deserialized such that half the packet (the most significant 16-bits) are sent to one collection while the other half is sent to the other collection. This entire process is shown in Figure 4.4.

Should the design be scaled further to include more than 8 collections, the streams can be time multiplexed so that more than two streams map to one port. This technique will work until we reach the maximum bandwidth of the DDR3 protocol running at the available frequency.

5. PERFORMANCE COMPARISON AND ANALYSIS

The nn-X implementation described in this work is prototyped on the Xilinx ZC706 platform (refer to Table 5.1).

The ZC706 development board contains two ARM Cortex-A9 cores, 1 GB of DDR3 memory and a programmable logic array. The programmable logic can fit up to 8 collections, each with one 10×10 convolution engine, one max-pooling module and one non-linear mapping module. We measured the power consumption of the entire board in this configuration to be 8 W and 3 W for the Zynq SoC and DDR3 memory. This large discrepancy exists because the ZC706 board contains many features that are not used by the design but cannot be turned off. These include a transceiver, a DDR3 SODIMM dedicated for the programmable logic, analog circuitry and I/O. However, since the power consumption of the board is measured while that of the relevant components is modeled, we use the larger number for sake of accuracy.

The ZC706 platform was chosen because performance increases linearly with the number of collections, and being able to fit 8 collections gave us a favorable balance of performance and performance per watt.

Table 5.1.: This table describes nn-X's hardware specifications

| | |
|------------------------------|-----------------------------|
| Platform | Xilinx ZC706 |
| Chip | Xilinx Zynq XC7Z045 SoC |
| Processor | 2 ARM Cortex-A9 @800 MHz |
| Programmable Logic | Kintex-7 |
| Memory | 1 GB DDR3 @533MHz |
| Memory bandwidth | 3.8 GB/s full-duplex |
| Accelerator frequency | 142 MHz |
| Number of Collections | 8 |
| Peak performance | 227 G-ops/s |
| Power consumption | 3 W (Zynq+mem), 8 W (board) |

Torch7 was used as the main software tool in this work [23]. Torch7 is a module implemented in the Lua programming language.

Performance analysis revolved around three design areas -

1. Performance with increasing resources - In an embedded system, resource utilization is key to delivering high performance. If there are long bubbles in the pipeline or some of the available resources remain unused for a significant amount of the processing time, then performance will be sub-optimal. As is, embedded systems have area constraints and utilizing all of the available resources is key to increasing performance.
2. Performance per unit power - This is an important metric because it defines whether a system is embeddable or not. A system with low performance per unit power consumed might not be desirable over a large system like a graphics processor due to the relative ease of programming of GPUs.
3. Raw performance - This metric is also important because delivering a large performance per unit power consumed is not enough if raw performance is too low. The right balance of power and performance needs to be achieved when targeting embedded systems.

The rest of this chapter analyzes these three areas of the nn-X system.

5.1 Performance per Resource

On the ZC706 prototype platform, nn-X features up to 8 collections. Since the Zynq device on the board has a maximum of four ports to memory, routing independent streams from each collection can be difficult. However, the concatenation mode described in Chapter 4, Section 4.2 mitigates this problem by efficiently routing all eight streams through the four ports. This prevents the nn-X coprocessor from stalling mid-stream.

Figure 5.1 shows the performance of the nn-X coprocessor with increasing collections. The coprocessor is within 89% of its theoretical maximum. The overhead exists due to

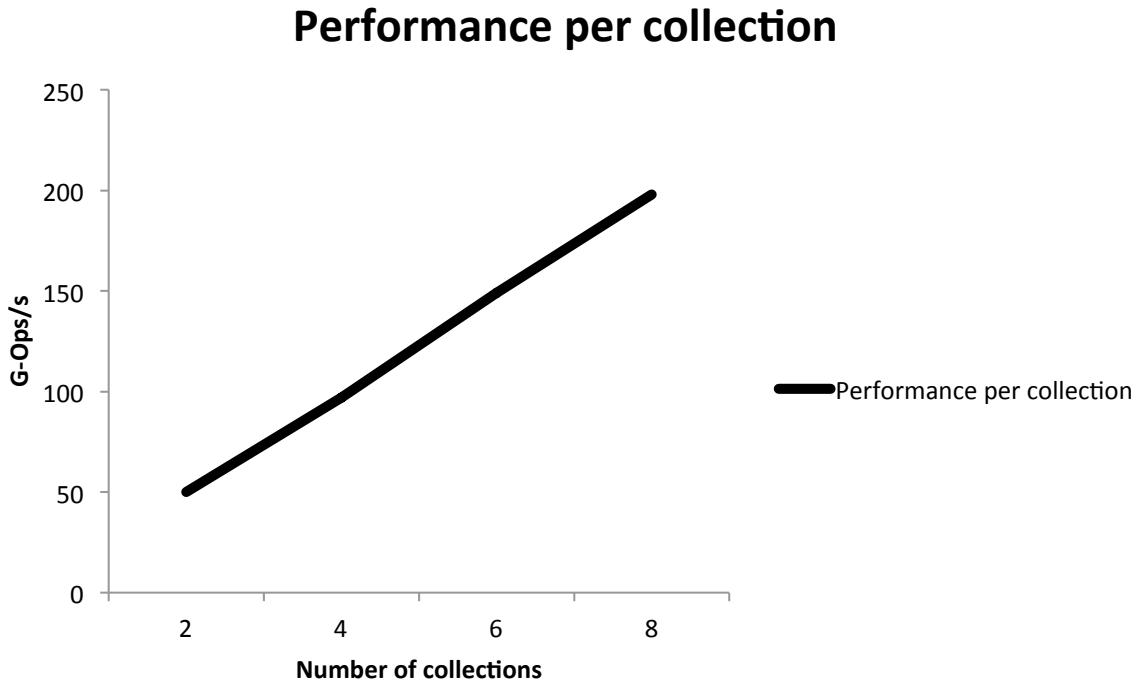


Fig. 5.1.: Performance of the nn-X coprocessor with increasing collections. The input was a 500×500 image to a 8 collections, each featuring 10×10 disparate convolution kernels.

the set up time of the DMA transaction. This demonstrates scalability, subject to availability of bandwidth. Each collection can process 270 megabytes of data per second. This translates to a bandwidth requirement of 2.11 gigabytes per second through the four ports. The maximum bandwidth available from the DDR3 protocol when running at 533 MHz is 4.2 gigabytes per second. This implies that the current architecture can scale up to 16 collections without bottlenecking the memory bus or requiring faster memory.

5.2 Performance Per Unit Power Consumed

For this metric, we compare the performance per unit power of the nn-X system to systems commonly used to process ConvNets. The results are shown in Figure 5.2. Most desktop and laptop CPUs and GPUs peaked at under 3 G-ops/s-W even when the algorithm was optimized to take advantage of hardware acceleration. Mobile processors reported better efficiencies of 8 G-ops/s-W.

nn-X (red) implemented in programmable logic was able to deliver more than 25 G-ops/s-W. nn-X's embeddable factor is six times that of the Snapdragon 800 SoC and twenty times that of NVIDIA's GTX 780. Figure 5.2 compares nn-X to custom processors running at much higher frequencies. An implementation of nn-X in silicon at similar process nodes would significantly improve its performance.

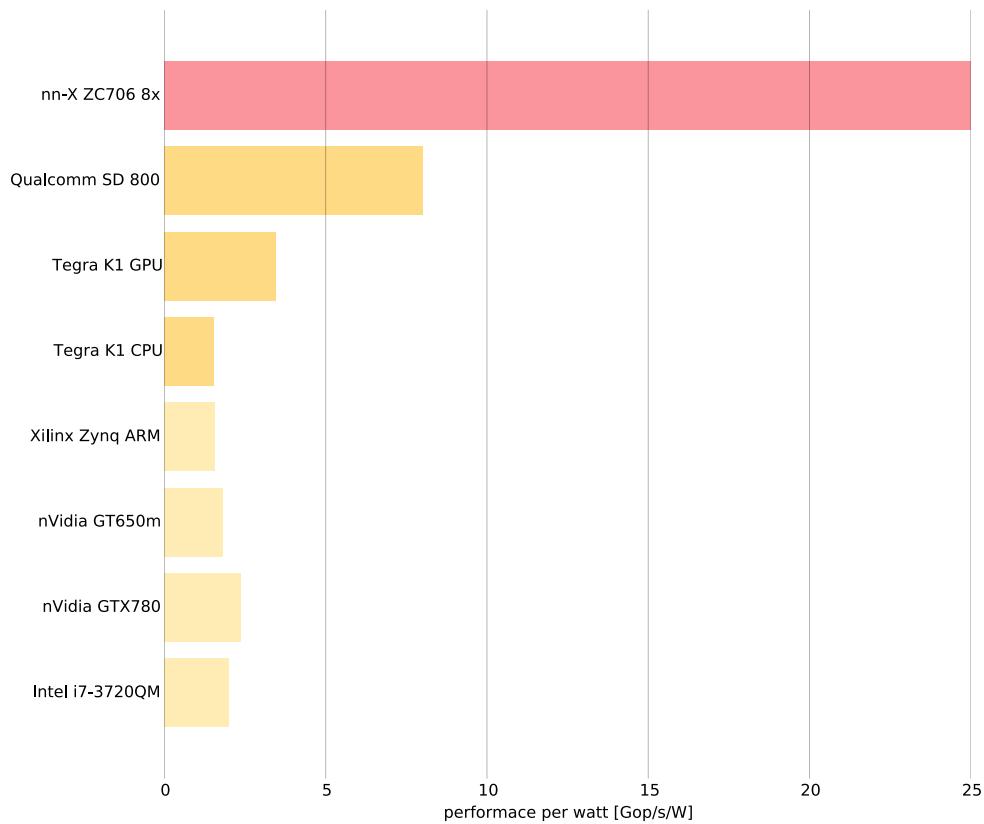


Fig. 5.2.: Performance per watt of different platforms. Most desktop CPUs and GPUs gave under 3 G-ops/s-W while mobile processors performed slightly better. nn-X (red) implemented in programmable logic was able to deliver more than 25 G-ops/s-W.



Fig. 5.3.: Single neural network layer with 4 input planes of 500×500 , 18 output planes and 3.6 billion operations per frame. nn-X computed one frame in 6.2 ms and was 271 times faster than the embedded processors.

5.3 Raw Performance

We developed demonstration applications for neural networks in Torch7 to benchmark the raw performance of the nn-X system.

Figure 5.3 shows an image from the first application: a fully-connected neural network layer with 4 inputs and 18 outputs. The network used 10×10 convolution kernels with 4×18 random filters, a max-pooling operation of 2×2 and thresholding. This network required 2.8 billion operations per frame. In this application, nn-X computed one frame in 11.2 ms and was 271 times faster than the embedded ARM processors. nn-X's measured performance was 200 G-ops/s, which is more than 83% of its theoretical peak performance.

The filter bank test is representative of the hidden layers in a ConvNet. While the first layer has either 1 or 3 inputs, the hidden layers can have several inputs, ranging from 16 to as many as 192 in the network designed by Krizhevsky et. al. in [4]. Figure 5.4 shows the performance across embedded devices.

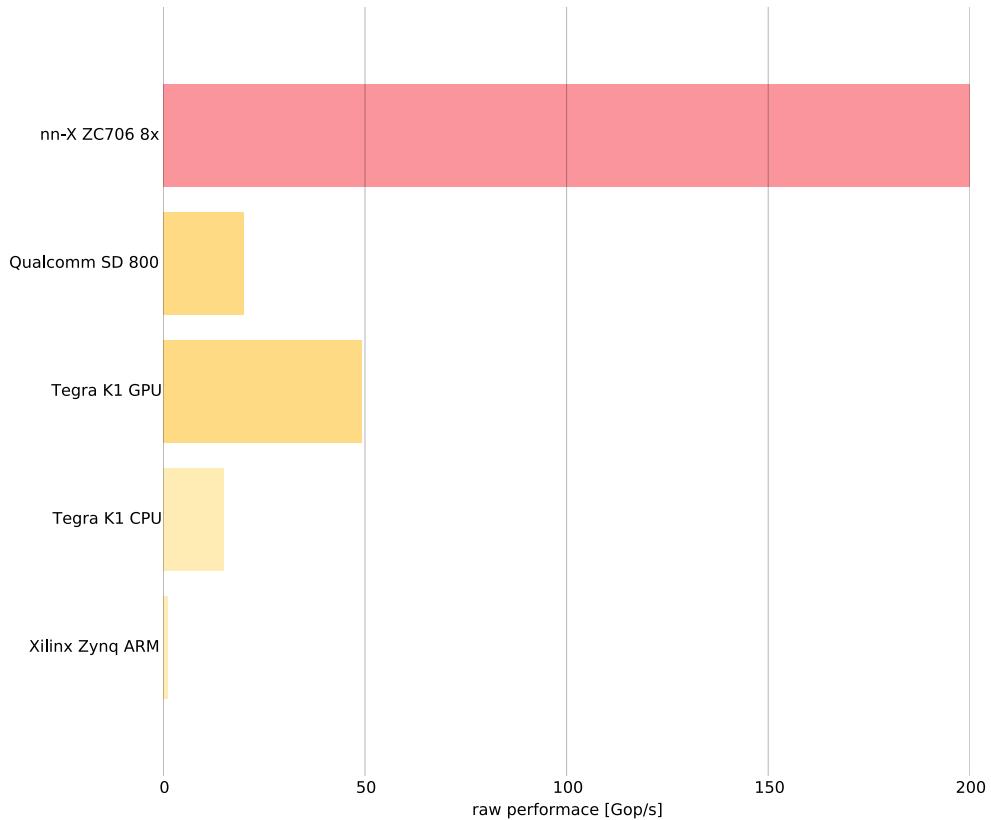


Fig. 5.4.: Performance comparison across embedded devices when running the filter-bank test.

In fact, nn-X is outperformed in raw performance only by NVIDIA’s GTX 780 GPU. The accelerator outperforms the Intel i7 mobile processor and the NVIDIA GT650m mobile GPU in raw performance by 2 and 4 times respectively. Those devices are not shown here due to them not being true *embedded* processors.

The next application is the face detector described by Farabet in [14]. We used a slightly modified version of this network. The first layer comprises 16 feature maps of 5×5 and is fully connected with the second convolution layer which comprises 64 feature maps of 7×7 . Each of these layers is interspersed with max-pooling of 4×4 and thresholding. The input to the network was a 500×350 greyscale image. This network requires 552 M-ops

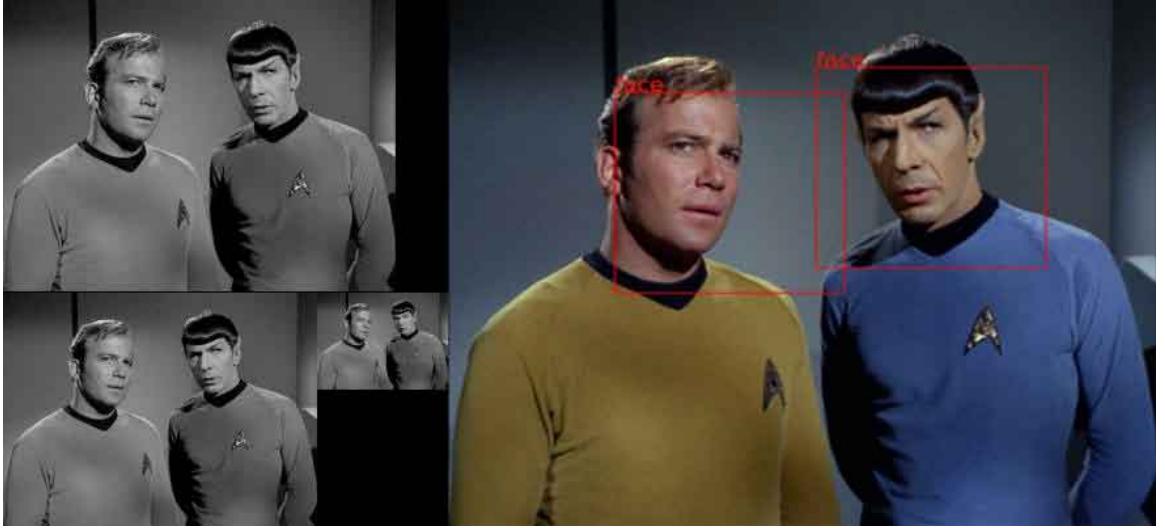


Fig. 5.5.: A face detector application with 552 M-ops / frame. nn-X was able to process a 500×350 video at 42 frames a second and was 115 times faster than the embedded processors. The image on the left is a multi-scale pyramid to provide scale-invariance to the input.

per frame and includes a multi-scale pyramid with scales of 0.3, 0.24, 0.1. Construction of this pyramid is a pre-processing step that is performed on the ARM processors. The pyramid itself is used for providing scale invariance to the input as the network is trained on images that are all the same size while inputs when running the feed forward network can be varying in size based on the distance of the user from the camera. The multi-scale input is then sent to the network for detection. nn-X was more than 115 times faster than the embedded ARM processors.

The raw performance of this network on the embedded devices is shown in Figure 5.6. The Tegra GPU is not able to perform as well as in the filterbank because of under-utilization of resources. While the performance of the GPU drops by a factor 10, nn-X suffers a drop by a factor of 2. This is primarily because the size of kernels is smaller than the maximum possible and this causes under-utilization of coprocessor resources.

The third application was a street scene parser capable of categorizing each pixel of the input image into one of eight categories: buildings, cars, grass, persons, road, street signs, sky and trees. This network requires 350 M-ops to process one frame.

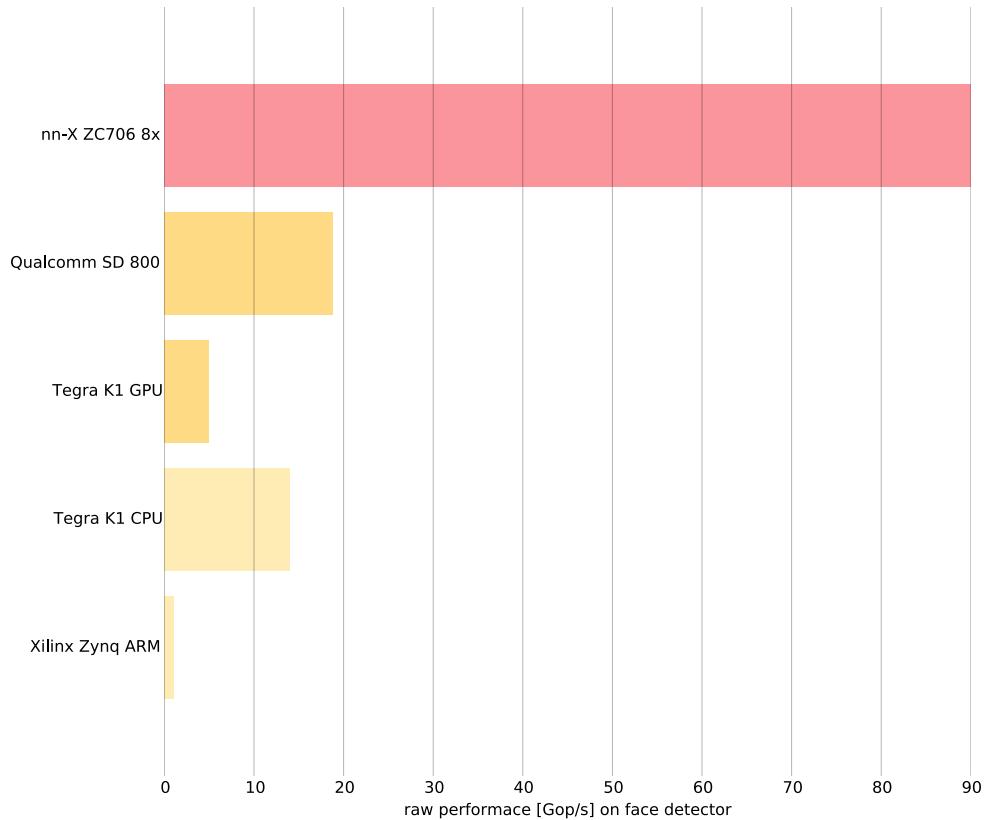


Fig. 5.6.: Comparison of performance across embedded devices. nn-X performs more than 4 times better than the next fastest processor on the face detector.

Figure 5.7 demonstrates nn-X performing full-scene understanding of a typical scene encountered when driving an automobile. nn-X processed a 510×288 video sequence in 4.5 ms, and was 112 times faster in processing time than the embedded ARM cores for this application.

The performance for the devices used is shown in Figure 5.8. In this application too, nn-X outperforms all other devices by at least a factor of 3.

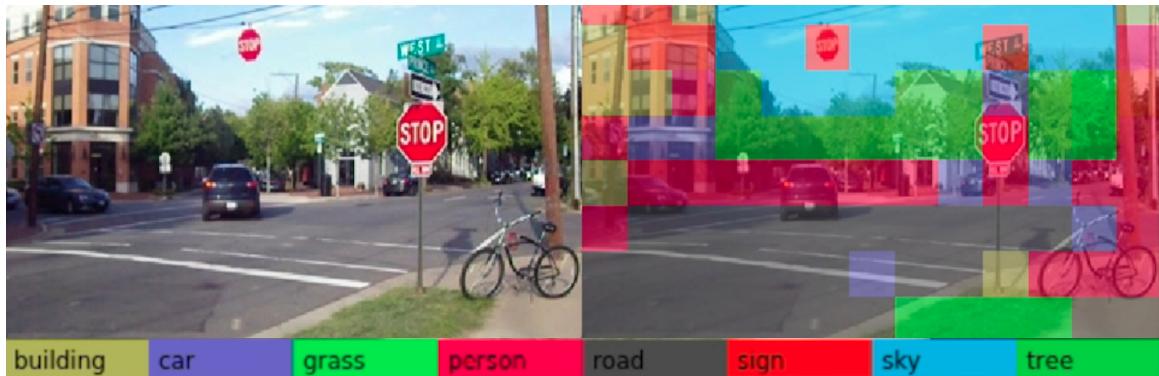


Fig. 5.7.: Street scene-parser requiring 350 M-ops / frame. This application processes a 510×288 input in 4.5 ms and produces an 8-class label for each frame. On this application, nn-X is 112 times faster than the Zynq processors.

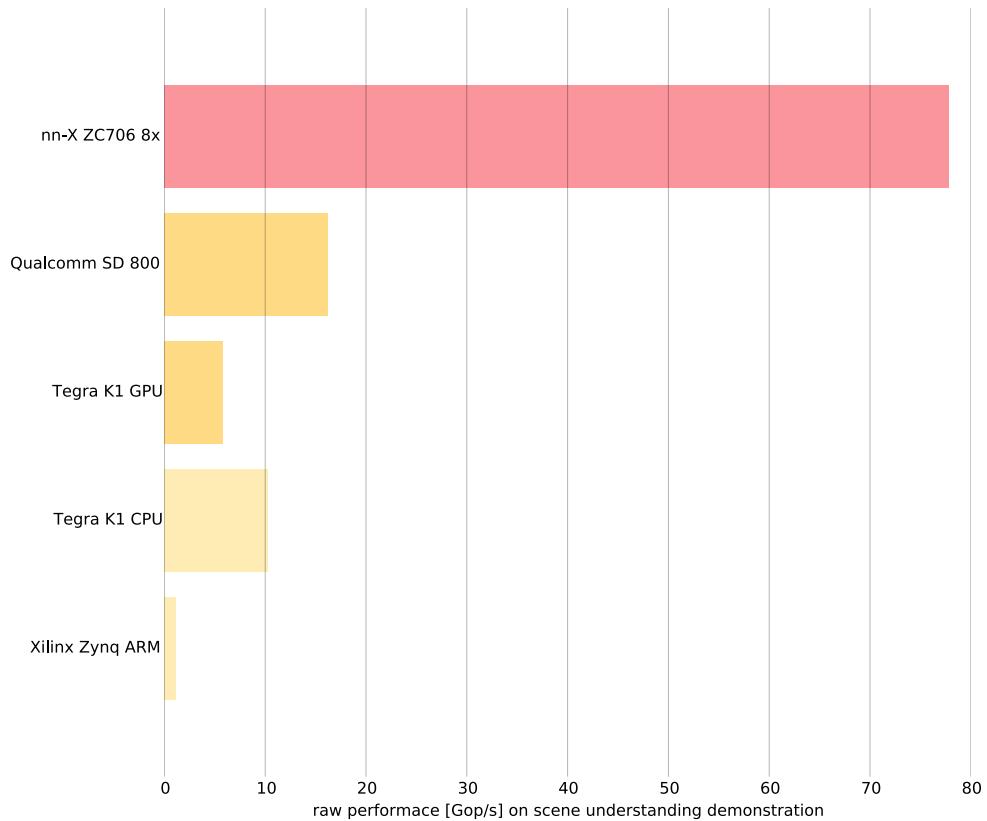


Fig. 5.8.: Comparison of performance across embedded devices. nn-X performs at least 3.5 times better than the embedded processors.

6. DISCUSSION

In this chapter we analyze the large performance benefit of this architecture over general purpose processors. One advantage is nn-X’s large parallelism; eight convolutional engines of 10×10 can deliver up to 227 G-ops/s while running at 142MHz. General purpose processors run at much higher frequencies. This chapter discusses two main strengths of the nn-X architecture.

6.1 Lack of Control Flow

The nn-X coprocessor does not process conditional statements. The compiler translates a network into the ConvNet’s operators. Lack of conditional statements allows nn-X to use every clock cycle to perform useful computation. In general purpose processors, conditional statements result in inefficient use of a processor’s pipeline [24].

In ConvNets, the convolution operator has few branches in its code as compared to the max-pooling and thresholding operators. Furthermore, branches in the convolution operator are mostly predictable on account of them being the result of iterative statements (“for” loops). However, the conditional statements in the max-pooling and thresholding operators are not predictable as their path is based on the value of the input pixel. This causes a performance drop in CPUs due to branch mispredictions. On GPUs, control divergence causes a drop in throughput [25].

To demonstrate this, we used two model deep networks. The first model consisted of an input layer of 3×16 kernels of 10×10 and an output layer of 16×32 kernels of 7×7 . The second model consisted of the same convolution layers but these were interspersed with max-pooling and thresholding operations. We used the same platforms from Figure 5.2 to perform this experiment.

In this context, we define efficiency as the performance achieved when running the model with only convolution layers versus the performance achieved when running the model with the max-pooling and threshold operations included. With the Torch7 package, all general purpose processors achieved an efficiency between 75% to 85%. nn-X achieved an efficiency close to 100%.

We explain this by the fact that in nn-X, the output of a convolution does not need to be written to memory due to the cascade of pipelined operators. Furthermore, as nn-X has no control flow, the output latency of the entire operation is simply equal to the combined latencies of each individual operator.

6.2 Efficiency of Memory Accesses

An efficient design for routing memory accesses is important for large-scale processing systems since such systems are often limited by memory bandwidth [26]. Processor caches are generally small compared to the total size of the inputs, intermediates and outputs [27]. This requires a processor to initiate frequent memory accesses for data that is not cached. On nn-X, the design of the local router within each collection combined with the memory router allows for maximum utilization of each collection. Each collection can produce one convolution output plane per memory access. With eight collections, eight results are produced with four memory accesses (which are done in parallel), provided all eight can be fed by data.

All collections can be guaranteed to be used because in a ConvNet because every input is required to pass through multiple filters before an output is produced and the number of filters is much larger than 8, which is the number of available collections. This guarantees that for most of the process, all collections will be in use. If the number of filters is not a multiple of eight, then in the last iteration, usage will not be 100%.

The convolution operation can be generalized as an N -to- M type as shown in figure 6.1. This results in two possible cases when running a feed-forward network. nn-X efficiently routes data in both cases.

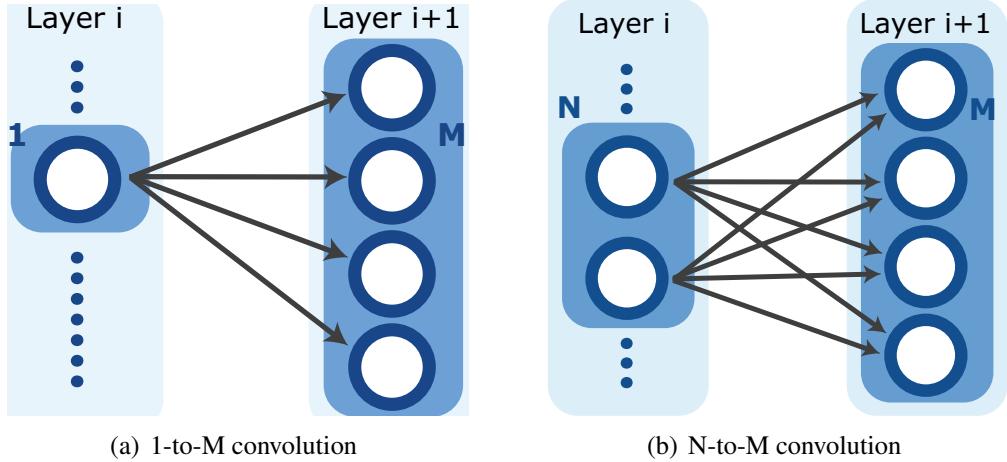


Fig. 6.1.: An example of 1-to- M and N -to- M convolution. These are the two most commonly encountered cases in ConvNets. A 2D diagram is used to visualize the relation between the convolution layers for simplicity. Circles indicate convolutional planes. M and N are 4 and 2 respectively.

6.2.1 $N = 1$ case

The $N = 1$ case, as demonstrated in figure 6.1(a), occurs when the network has one input stream and M filters. This is typically found in the first layer of a ConvNet when using a greyscale input image. Such a layer would produce M outputs and no intermediate results. In nn-X, one input stream is routed to multiple collections by the memory router. It is then processed by the collection's operators before being routed back to the memory router as one output of the current layer. The memory router then sends this output to memory where it awaits its turn to be sent back in as an input to the next layer. This process is repeated with different kernels in different collections to produce multiple outputs in parallel as the input is routed to all eight collections at the same time. The outputs are concatenated so that they can be sent out through the four HP ports without stalling any collection.

For this case, we need only $\lceil \frac{M}{C} \rceil$ memory accesses where M is the number of kernels and C is the number of collections. In a general purpose processor, this would result in $3 \times M$ memory accesses, one access for each operator. Input images are generally large and cannot fit in first level caches of general purpose processors. Usually they get loaded into

the larger last level cache. While access to these caches is much faster than access to main memory, it is still on the order of tens of clock cycles, depending on the architecture [28]. This results in performance that is not significantly lower than peak performance but is not optimum either.

6.2.2 $N > 1$ case

The $N > 1$ case is typical of hidden layers where the multiple outputs generated by the first layer are sent in as inputs. This case is shown in figure 6.1(b). In this case, the current layer has N inputs, M outputs and $N \times M$ filter kernels as shown in figure 6.1(b). N 2D convolutions and their pixel-wise summation is required to generate one output.

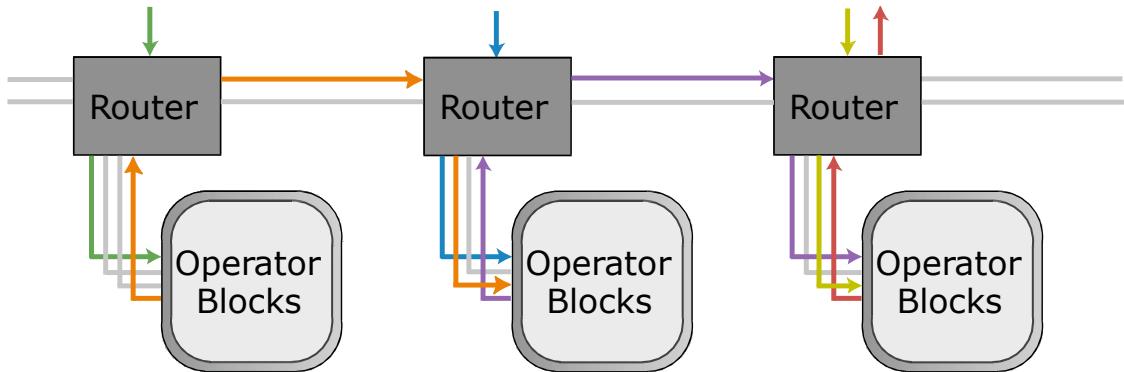


Fig. 6.2.: An example routing of a network producing one output from three inputs. The input to each router comes from the memory router. The output of the left-most operator block is a convolution (orange). It gets sent to the center collection where it is combined with the convolution of the blue input stream to produce the purple stream as another intermediate. This stream gets sent to the collection on the right to get combined with the convolution of the yellow input to produce the red result as the final output which is then sent to memory. The connections between neighbors facilitate rapid data transfer between collections.

Each collection has two bi-directional connections with its east and west neighbors. These connections, illustrated in Figure 6.2, enable a collection to send a stream to either of its neighbors without interrupting the memory router or the collection's operators. As long as a neighboring collection's operator is available, the intermediate result produced does not

need to be sent back to memory. It can be combined with another intermediate, effectively reducing memory accesses by a factor of four (two intermediates being transferred back and forth between memory and coprocessor).

In contrast, general purpose processors' performance suffers greatly in this step because of the sheer volume of intermediates produced. As all intermediates will not fit in any of the on-chip caches, only some intermediates are cached. This results in frequent cache misses and memory accesses which causes significant drop in processor performance. For example, a 256×256 intermediate is 256 kilobytes in size when represented in IEEE 754 floating point numbers and a ConvNet's hidden layers can produce over a hundred such intermediates. In contrast, last level caches of even high performance server processors are a few tens of megabytes in size.

7. FUTURE WORK AND CONCLUSION

7.1 Future Work

Future work on the nn-X accelerator will revolve around two design points. The performance of the accelerator increases linearly with frequency. This is due to the pipelined nature of the operators which produce one output per clock cycle. If the clock frequency was higher, the throughput would increase. This can be achieved by running the operators at a frequency faster than that of the rest of the coprocessor. As long as the operators are fed with data, the accelerator throughput will increase.

The primary means for doing this is by increasing the memory bus width to 64-bits. Since the coprocessor uses 16-bit data words, this allows for both concatenation and packing of data. With a faster clock, data from the operators is available faster than it can be sent to memory. Two of these data words can be packed into one 32-bit data word. Then, two such 32-bit packets from two collections can be concatenated together to form one output packet which is sent to memory. This will work provided the operators are clocked at a frequency that is at most twice of the frequency at which the packer is clocked. Above this, there will be no benefit because the outputs will not be removed fast enough and would require the operators to stall which effectively reduces their frequency to twice the frequency of the packer. Since power consumption linearly increases with clock frequency, the performance per watt metric of this new architecture would need to be tested to ensure that it is not lower than the architecture described in this document.

The second area for improvement is within the convolution engine. Currently, regardless of the size of the convolution kernel, the engine can only perform convolution with a single kernel. For example, a convolution engine with a maximum possible convolution of size of 10×10 requires 100 multiple-accumulate (MAC) units. However, if such a convolution engine is performing a 5×5 convolution, only 25 of those MAC units are used. This

will result in a performance that is only 25% of the theoretical maximum, while consuming just as much power.

These other MAC units can be used if 4 such 5×5 convolutions can be performed in one convolution engine. This would require the MAC units to be able to be used globally as one kernel or be broken down into smaller units. This presents the challenge that one convolution produces one stream. Multiple convolutions in one collection will produce multiple streams which would need to be sent efficiently to memory without stalling any other resource (or itself) or be stored in an on-chip cache.

7.2 Conclusion

In this thesis, we presented the case for designing a specialized hardware architecture targeted to accelerating convolutional neural networks (ConvNets). ConvNets are used in synthetic vision systems because of their versatility and as such, are suitable for a variety of vision tasks. These models are inherently parallel and can be accelerated on custom data-parallel architectures to give a low powered mobile system capable of achieving high performance.

We then presented the nn-X architecture. nn-X is a low powered coprocessor that can be installed alongside any mobile system. The architecture comprises the three main mathematical operators used in ConvNets - the two dimensional convolution operator, the max-pooling operator and the non-linear operator. These operators are bundled together into processing elements called collections. An efficient routing network is used to deliver high performance by optimally utilizing the available hardware resources. We also described the supporting hardware used to complete the nn-X architecture and interface it with the host processor.

Finally, we demonstrated the performance of nn-X on a single-layer neural network, a network trained to detect faces and a network understanding objects in a road scene. nn-X was faster than embedded processing systems in all applications while consuming significantly lower power. The system was prototyped on a programmable logic array using the

Xilinx Zynq ZC706 platform. nn-X was tested to be at least 3.5 times faster than other embedded solutions in performance per unit power and over 3 times faster in raw performance. These results make nn-X an excellent candidate as an embedded convolutional neural network accelerator.

LIST OF REFERENCES

LIST OF REFERENCES

- [1] S. Lazebnik, C. Schmid, and J. Ponce, “Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories,” in *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, vol. 2, 2006, pp. 2169–2178.
- [2] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, “Speeded-up robust features (surf),” *Comput. Vis. Underst.*, vol. 110, no. 3, pp. 346–359, Jun. 2008.
- [3] T. Serre, L. Wolf, S. Bileschi, M. Riesenhuber, and T. Poggio, “Robust object recognition with cortex-like mechanisms,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 29, no. 3, pp. 411–426, 2007.
- [4] A. Krizhevsky, I. Sutskever, and G. Hinton, “ImageNet classification with deep convolutional neural networks,” *Advances in Neural Information Processing Systems*, vol. 25, 2012.
- [5] R. Socher, B. Huval, B. Bath, C. D. Manning, and A. Ng, “Convolutional-recursive deep learning for 3d object classification,” in *Advances in Neural Information Processing Systems*, 2012, pp. 665–673.
- [6] D. Ciresan, U. Meier, and J. Schmidhuber, “Multi-column deep neural networks for image classification,” in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, 2012, p. 36423649.
- [7] J. Jin, A. Dundar, J. Bates, C. Farabet, and E. Culurciello, “Tracking with deep neural networks,” in *Information Sciences and Systems (CISS), 2013 47th Annual Conference on*, March 2013, pp. 1–5.
- [8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [9] A. Mohamed, G. Dahl, and G. Hinton, “Acoustic modeling using deep belief networks,” *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 20, no. 1, pp. 14–22, Jan 2012.
- [10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” in *NIPS Deep Learning Workshop*, 2013.
- [11] *Deep Learning*, 2013 (accessed May 20, 2014), <http://deeplearning.net>.
- [12] Q. Z. Wu, Y. Le Cun, L. Jackel, and B.-S. Jeng, “On-line recognition of limited-vocabulary chinese character using multiple convolutional neural networks,” in *Circuits and Systems, 1993., ISCAS '93, 1993 IEEE International Symposium on*, May 1993, pp. 2435–2438 vol.4.

- [13] S. Rajasekaran and G. Pai, *Neural Networks, Fuzzy Logic and Genetic Algorithm: Synthesis and Applications (with CD)*. PHI Learning, 2003. [Online]. Available: <http://books.google.com/books?id=bVbj9nhvHd4C>
- [14] C. Farabet, C. Poulet, and Y. LeCun, “An fpga-based stream processor for embedded real-time vision with convolutional networks,” in *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, Sept 2009, pp. 878–885.
- [15] J. Cloutier, E. Cosatto, S. Pigeon, F.-R. Boyer, and P. Simard, “Vip: an fpga-based processor for image processing and neural networks,” in *Microelectronics for Neural Networks, 1996., Proceedings of Fifth International Conference on*, 1996, pp. 330–336.
- [16] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, “Programmable stream processors,” *IEEE Computer*, pp. 54–62, Aug. 2003.
- [17] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2006. [Online]. Available: <http://books.google.com/books?id=57UIPoLt3tkC>
- [18] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, “Neuflow: A runtime reconfigurable dataflow processor for vision,” in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, 2011, p. 109116.
- [19] P.-H. Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun, and E. Culurciello, “Neuflow: dataflow vision processing system-on-a-chip,” in *Circuits and Systems (MWS-CAS), 2012 IEEE 55th International Midwest Symposium on*. IEEE, 2012, pp. 1044–1047.
- [20] H. P. Graf, S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, and S. Chakradhar, “A massively parallel digital learning processor,” in *Advances in Neural Information Processing Systems 21*, D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, Eds., 2009, pp. 529–536.
- [21] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’14. New York, NY, USA: ACM, 2014, pp. 269–284. [Online]. Available: <http://doi.acm.org/10.1145/2541940.2541967>
- [22] J. Holi and J.-N. Hwang, “Finite precision error analysis of neural network hardware implementations,” *Computers, IEEE Transactions on*, vol. 42, no. 3, pp. 281–290, Mar 1993.
- [23] R. Collobert, K. Kavukcuoglu, and C. Farabet, “Torch7: A matlab-like environment for machine learning,” in *BigLearn, NIPS Workshop*, 2011.
- [24] R. Nath, S. Tomov, and J. Dongarra, “Accelerating gpu kernels for dense linear algebra,” in *High Performance Computing for Computational Science VECPAR 2010*, ser. Lecture Notes in Computer Science, J. Palma, M. Dayd, O. Marques, and J. Lopes, Eds. Springer Berlin Heidelberg, 2011, vol. 6449, pp. 83–92. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19328-6_10

- [25] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, “Gklee: Concolic verification and test generation for gpus,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’12. New York, NY, USA: ACM, 2012, pp. 215–224. [Online]. Available: <http://doi.acm.org/10.1145/2145816.2145844>
- [26] M. Peemen, A. Setio, B. Mesman, and H. Corporaal, “Memory-centric accelerator design for convolutional neural networks,” in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, Oct 2013, pp. 13–19.
- [27] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber, “Deep, big, simple neural nets for handwritten digit recognition,” *Neural computation*, vol. 22, no. 12, pp. 3207–3220, 2010.
- [28] F. Sleiman, R. Dreslinski, and T. Wenisch, “Embedded way prediction for last-level caches,” in *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, Sept 2012, pp. 167–174.