Embedded Streaming Deep Neural Networks Accelerator with Applications

Aysegul Dundar, Jonghoon Jin, Berin Martini and Eugenio Culurciello, *Senior, IEEE*

***Abstract*—Deep Convolutional Neural Networks (DCNNs) have become a very powerful tool in visual perception. DCNNs have applications in autonomous robots, security systems, mobile phones, and automobiles where high throughput of the feed-forward evaluation phase and power efficiency are important. Because of this increased usage, many FPGA-based accelerators have been proposed. In this paper, we present an optimized streaming method for DCNNs' hardware accelerator on an embedded platform. The streaming method acts as a compiler, transforming a high-level representation of DCNNs into operation codes to execute applications in a hardware accelerator. The proposed method utilizes maximum computational resources available based on a novel scheduled routing topology that combines data reuse and data concatenation. It is tested with a hardware accelerator implemented on the Xilinx Kintex-7 XC7K325T FPGA. The system fully explores weight level and node level parallelizations of DCNNs and achieves a peak performance of 247 G-ops while consuming less than 4 watts of power. We test our system with applications on object classification and object detection in real-world scenarios. Our results indicate high performance efficiency, outperforming all other presented platforms while running these applications.**

***Index Terms*—Embedded Systems, Deep Neural Networks, Applications of Deep Neural Networks**

I. Introduction

ARTIFICIAL VISION SYSTEMS aim to provide visual understanding from raw images, i.e., convert high dimensional data like images and videos into useful low-dimensional data, where decisions can be made. There has been extensive research on such systems for the past two decades. These systems range from fully trained Deep Convolutional Neural Networks (DCNNs) to SIFT and SURF feature extractors [1], [2] and hierarchical models of the visual cortex [3]. In recent years, work on DCNNs [4]–[6] has shown promising results on visual understanding.

DCNNs are powerful tools that use many layers and filters to extract meaningful features of objects in images in a hierarchical manner. They consist of multiple layers of convolutions, each comprising between tens and hundreds of filters. Figure 1 describes the most common DCNN structure where each convolution is followed by a pooling (usually max-pooling) and a non-linearity operation (usually a rectified linear unit, a.k.a. ReLU [4]).

The first convolution layer extracts simple features like edges and curves from images. The pooling operation that directly follows the convolution layer provides scale invariance and makes the network more tolerant to distortion in images; however, the exact locations of the features are lost in the process. The convolution operations in the subsequent layers extract more complex shapes [7] because they are extracting features from the output maps of previous layers.

Recent advances in optimizing deep networks such as dropout [8], sub-sampling [9], [10] and efficient GPU implementations for training have enabled researchers to increase the scale of DCNNs with more layers and parameters that were not previously practical to train. This development has resulted in increased accuracy of many tasks, including object classification and detection [4]. As a result, DCNNs have become particularly useful for applications in autonomous robots, security systems, mobile phones, and automobiles, which require real-time execution and high accuracy. DCNNs are, however, computationally very expensive due to the fact that over ninety percent of the time required for convolution operations with images is spent on processing by the filters [11].

GPUs are becoming a common alternative to custom hardware in vision applications because they are inexpensive and easily programmable [12], [13]. In particular, while training DCNNs, GPUs provide high-speed processing of hundreds of images at the same time. However, custom hardware can provide better performance during the feed-forward prediction phase with less power consumption, which is necessary for embedded systems. By developing a custom architecture and control method adapted to DCNNs, the product of power consumption by performance can be improved. Because of these advantages, extensive research has been devoted to custom architectures for convolutional networks or related algorithms [14]–[20].

This study presents a highly optimized control method for the scalable hardware architecture for DCNNs. The control method acts as a compiler and transforms high-level representations of DCNNs into operation codes for the purpose of executing applications within a hardware accelerator. The control method takes advantage of the characteristics of DCNNs and explores the computational power of the custom hardware. We demonstrate real-time applications of this system with classification and detection of objects in videos.

The results of this study are organized as follows: Section II explains related work. Section III describes the architecture of the custom hardware. The strengths and limitations of the custom hardware are presented in this section. Section IV describes our control method that is optimized for DCNNs and custom hardware. Section V gives experimental results on

Aysegul Dundar, Berin Martini and Eugenio Culurciello are with the Weldon School of Biomedical Engineering, Purdue University, West Lafayette, IN 47907 USA.

Jonghoon Jin is with Electrical Engineering, Purdue University, West Lafayette, IN 47907 USA

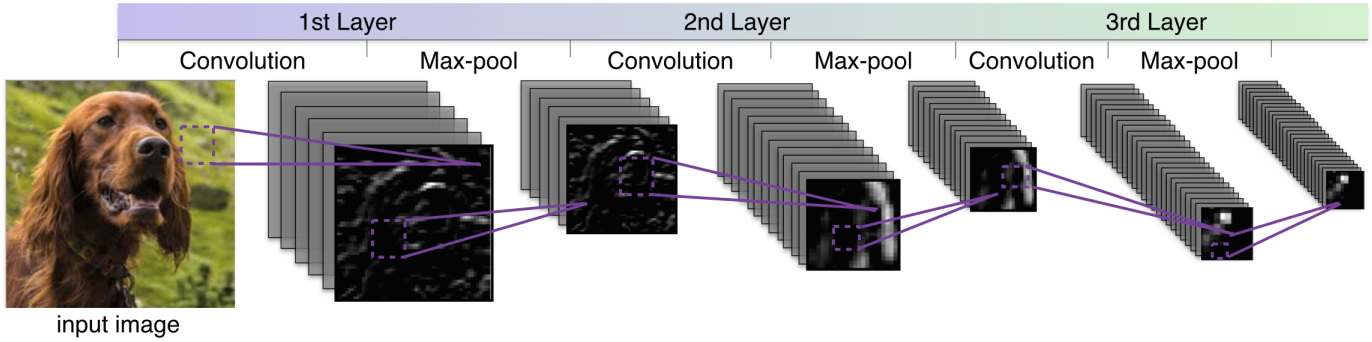


Fig. 1. Diagram of a Deep Convolutional Neural Network. It includes n-layer (blocks) of three consecutive operations: convolution, max-pooling and non-linearity (ReLU). The last layer is usually a linear classifier that is followed by softmax regressor which translates the set of inputs into a set of probability distributions which can be the labels of objects contained in an input image.

the performance of the system. Section VI describes two real-world applications of DCNNs, classification, and detection of objects in videos. Training and network details are described in this section. The final section is a comprehensive summary.

II. Related Work

With the recent success of Deep Convolutional Neural Networks (DCNNs) in object recognition, there has been extensive interest in accelerating forward and backward data processing of DCNNs. Parallelism is explored at many levels, including weight parallelism (parallel sum of the products of computation in convolution) and node parallelism (computation over multiple convolutional planes) [21]. Because the training phase involves the processing of many input images before a weight update, parallelization across images is also explored. Note that in real-time applications this acceleration is not possible since the input images arrive in regular time intervals and an output is expected immediately after each image is sent.

Regarding CPUs, there have been highly optimized implementations of DCNNs [22], [23] that explore cross image parallelization with multi-threading and implementing convolutions as highly optimized matrix-matrix multiplications (using BLAS). Other approaches are proposed, such as the unrolling of convolution operations (multiple nested for loops) for cache-friendly memory access [24], and the linear quantization of network weights and inputs [25]. Despite the benefits of such implementations, DCNNs on general purpose processors are still too demanding, power-consuming, and slow to be used for real-time applications in an embedded system environment.

While GPUs are gaining popularity in the training phase, they are not commonly present in embedded systems. The challenges in running DCNNs on GPUs are mostly centered on memory and communication. Different communication methods for GPUs have been explored for the purpose of overcoming the bandwidth limitation between the host system and GPU memory [13], [26]–[28]. The small local memory spaces in the GPU limit the ability to properly store the images, as do the filter coefficients of large networks [29]. Other optimizations take advantage of processing multiple inputs for training in a mini batch mode [12]. However, this is not practical for real-time applications where there is a continuous feed of input, and outputs are expected immediately.

Despite the fact that GPUs are becoming a common alternative to custom hardware, custom hardware can provide better performance with significantly less power consumption, which is necessary for mobile, embedded platforms. By developing a custom architecture and control method fully adapted to DCNNs, the product of power consumption by performance can be improved enormously. Because of this, there has been extensive interest in utilizing custom hardware to accelerate convolutional neural networks [30]–[35]. Custom hardware can uncover massive parallelism compared to CPUs. It also consumes low power, unlike GPUs. This advantage comes from a custom implementation where a large number of logic units, specialized to operations required for DCNNs, can operate in parallel.

Node parallelism provides a significant performance benefit for DCNNs [21], [30], [33] based on the fact that each convolutional plane is independent of others in the same layer. The degree of parallelism can be increased by placing as many processing units as silicon can hold. However, the actual performance is often limited by its memory bandwidth during DCNN computation. Since the number of connections to produce a single output plane is much higher than the number of processing units, node parallelism necessarily generates intermediate results that need to be stored in memory. Such intermediates require frequent memory access between a host processor and memory. This causes significant overhead time when used for large-scale neural networks.

Efficient memory and operation scheduling, as well as unit parallelism, are crucial factors in DCNN acceleration. Recently, an FPGA-based accelerator for DCNNs was proposed, containing an analytical design scheme using roofline model [36]. This method uses floating point operations in comparison to our study, which uses fixed point operations. Overall, our method achieves better peak performance in all of the experiments we tested. Furthermore, our method is capable of running the whole network except the linear layer, as opposed to only convolutional layers.

In this study, we propose a routing scheme that combines data reuse and concatenation to reduce the memory access limitation of custom hardware. This routing scheme enables

maximum node level parallelization of DCNNs. Whereas the hardware fully explores the weight parallelism, the control method enables the full node parallelism with available resources in the hardware accelerator. Our control method acts as a compiler by transforming a high-level representation of DCNNs into operation codes to execute applications in a hardware accelerator. We discuss different optimizations and improvements in our control method while running DCNNs. This study is in line with previous work [37]–[39] but includes many improvements and an in-depth explanation of our methods. The initial study [37] was based on a small FPGA; therefore, the limitation was on computational resources rather than the memory access bandwidth. In other words, there was no need for an optimized routing scheme. The study presented herein [38] focuses on the convolution engine rather than the control method that runs the applications. While the recent study [39] investigates memory optimizations, this method is not generic and does not utilize the resources after the first iteration because of the intermediate values occupying the bandwidth. In our work, we present a full system that combines data reuse and concatenation, and we present object classification and detection applications that run on our system.

III. Hardware Accelerator

In this section, we present the architecture of the hardware accelerator for running DCNNs in the feed-forward prediction phase. The capabilities and limitations of this hardware were the most important considerations when we implemented our control method. The hardware system is implemented on the Xilinx Kintex-7 XC7K325T FPGA.

A. Architecture

The architecture of the hardware accelerator is based on [37] where the hardware is divided into two main areas: the memory router and collections that include the operators for processing images. Hardware operators are bundled together into a unit called a *collection*. All the operators and routers can be configured at run time and each module operates independently. In other words, configuring one module does not require stalling any others. All operators use the Q8.8 number format - which has been tested to provide virtually identical results to 32-bit floating number format - for the feed-forward phase computation, even though the DCNN has been trained in 32-bit floating number format. These two main parts, memory router and collections, are explained in the subsections below.

1) Memory Router: The memory router interfaces with eight high-throughput I/O ports and four collections. It acts as a gateway to the accelerator and directly communicates with AXI bus. It is implemented as a crossbar switch where all incoming and outgoing streams pass through this module. Having a larger crossbar switch could ease the data traffic in the memory router. However, it would also cause inefficient use of logic area due to the dense grid pipeline. In order to meet these area constraints, only essential streams are routed to the memory router while intermediate streams are delivered to the neighboring collections. The memory router can route incoming data streams to one or more outputs based on the configured routing topology. Moreover, one part of the router can be dynamically reconfigured without halting the functionality of the rest.

2) Collections: Hardware operators are bundled together into a single module called a *collection*. Each collection contains a convolution, a max-pooling, a non-linearity module, and a stream-adder. The presented system can hold four collections. Each convolution operation in a collection can be configured to do 1 - 12×12 convolution or smaller, 4 - 6×6 convolution or smaller, 16 - 3×3 convolution or smaller, and so on.

Inputs and outputs appear as data streams for all modules. The output from the convolution operator can be streamed into the max-pooling or non-linearity operator within the same collection. Producing output maps for 3D convolutions requires the accumulation of element-wise sums over multiple 2D convolved maps. To achieve this, the output of the convolution from one collection can be streamed into the second collection where it can be summed together with the output of the convolution from the second collection. This process can be repeated among all collections.

B. High-throughput Data Ports

The FPGA has eight high-performance ports to DDR3 memory. These high-performance ports tap into DDR3 memory using the AXI buses. Each AXI DMA is bidirectional and can transfer 128-bit data words per clock cycle per direction.

DCNNs process hundreds of images with different filters and produce hundreds of intermediate results. Therefore, the bandwidth of the system can be a significant limitation in data transfer. Since the presented system has four collections, each capable of processing images with multiple filters, enough bandwidth could stream different images to each convolution operator in each collection and sum all of them together in one transaction. However, having only eight AXI ports to stream data in and out of the hardware accelerator proves to be a prominent handicap. It is not possible to send unique streams of data from memory to each convolution engine. Our method presents a novel routing scheme to achieve the full utilization, as described in subsection IV-B.

C. Computational Resources

The operations used by DCNNs are implemented in the custom hardware as follows:

1) Convolver: The convolution operation is the basis of DCNNs. Convolution with trained filters is used to extract useful features from the input images or from the output of the previous layers, in which case convolution extracts more complex features.

Weights in DCNNs can be described as 4-dimensional filters: $W \in \mathbf{R}^{C \times X \times Y \times F}$, where C is the number of input channels, X and Y are the spatial dimensions of the filter, and F is the number of filters or the number of output channels. The output maps of the convolution operation are called feature maps. The number of feature maps is equal

to the number of filters. For each feature map, inputs are convolved with a filter $W \in \mathbf{R}^{C\times X\times Y}$ and are described as:

$$F_f(x,y) = I * W_f = \sum_{c=1}^{C}\sum_{x'=1}^{X}\sum_{y'=1}^{Y} I(c, x-x', y-y')W_f(c,x',y') \quad (1)$$

assuming a stride of one where f is an index of the feature maps, $I \in \mathbf{R}^{C\times N\times M\times F}$ is the input map, and N and M are the spatial dimensions of the input.

$$F_f(c,x,y) = \sum_{x'=1}^{X}\sum_{y'=1}^{Y} I(c, x-x', y-y')W_f(c,x',y') \quad (2)$$

The kernel is pre-loaded from memory and cached for the duration of the convolution. As an image is streamed in, one input pixel results in one output pixel. This does not include an initial set up delay that occurs due to the pipelined nature of the hardware.

2) Max-pooler: In the max-pooling operator, the images are divided into a set of squares $p \times p$, and for each square, the maximum value is outputted. This operation in practice is a non-linear down-sampling by a factor of p. It gives strength to DCNNs by providing a form of translation invariance. Furthermore, it reduces computations for the upper layers.

3) Non-linearity operator: In DCNNs, a non-linearity operation usually follows either convolution or max-pooling operations. Non-linearity operations increase the capabilities of DCNNs to separate high-dimensional inputs. Without a non-linearity, a composition of linear functions is itself a linear function, which has limited power to encode information.

The Rectified Linear Unit (ReLU), $f(x) = \max(0; x)$, is one of the most widely used non-linearity operations [4]. The ReLU operation in the custom hardware produces one output per clock cycle.

4) Stream Adder: This module computes an element-wise addition of two incoming streams and produces a single stream as output. It can process element-wise arithmetics, and its main role in the context of DCNNs is to complete the 3D convolutions by summing up multiple 2D convolution maps from the convolution engines. This operator takes one stream from the neighbor collection and performs operations with the other stream that comes to the collection that contains it. The arithmetic occupies a single DSP slice but effectively processes the accumulation of all elements in the plane at one clock cycle when used along with other pipelined processing units.

IV. Control Method

This hardware accelerator requires special software to interpret high-level DCNN abstraction into a sequence of control instructions for configuring the connections in the accelerator. This software, the control method, acts as a compiler and transforms high-level representations of DCNNs into operation codes to execute applications within a hardware accelerator. The method takes sequential descriptions of networks from the Torch environment [40] and parses them to exploit different levels of parallelisms [21] and optimizations.

There are three main features of the control method that significantly improve performance:

- *Across modules optimizations:* Max-pooling and non-linearity operations generally follow a convolution operation. These three operations can be cascaded. As the input streams in, the output of the layer can be produced in a pipelined manner. This concatenation part takes place in the *network parser* and will be described in section IV-A.
- *Across images, within a module:* Operations that are independent of each other can be parallelized. However, each independent operation requires its own input and output streams, and the bandwidth of systems becomes a major bottleneck that prevents efficient energy flow. We propose a novel routing which combines data reuse and data concatenation for maximum parallelization. This part is described in section IV-B.
- *Smart configuration:* For videos or a collection of images, the same network is run multiple times on different images. The sequence of codes for configuration is created only in the beginning and is serialized for memory-optimized access. Caching these sequences of codes in hardware further boosts the performance. This process takes place in the *configurator* and will be described in section IV-C.

The hierarchy of our control method, from top down, includes: network parser, resource allocator, and finally hardware configurator. Each part of the control method is explained below.

A. Network Parser

The network parser explores across-module optimizations while creating a list of operations needed to execute a given DCNN. The network parser scans the sequential description of DCNNs (Torch network definition [40]) and combines the operations that can be calculated in one collection in the custom hardware. Across-module optimizations are obtained by cascading operations such as convolution, max-pooling, and non-linearity. These operations in DCNNs generally follow each other; hence, these three operations can be cascaded (i.e., the stream of the output of convolution is fed to the max-pooling operation and the output of that is fed to the non-linearity operation). In this process, input and output streams do not need to be sent to the memory router between each operation. In this step, the network parser does not use any low-level information, and is therefore independent of available resources in the custom hardware. After the list is created, it is analyzed in the *resource allocator* to run as many of the operations in parallel as possible based on the resources.

B. Resource Allocator

The resource allocator manages the hardware resources of the coprocessor and assigns queued operations from the network parser into the collections (computing units). The routing topology is drawn by the allocator in order to maximize

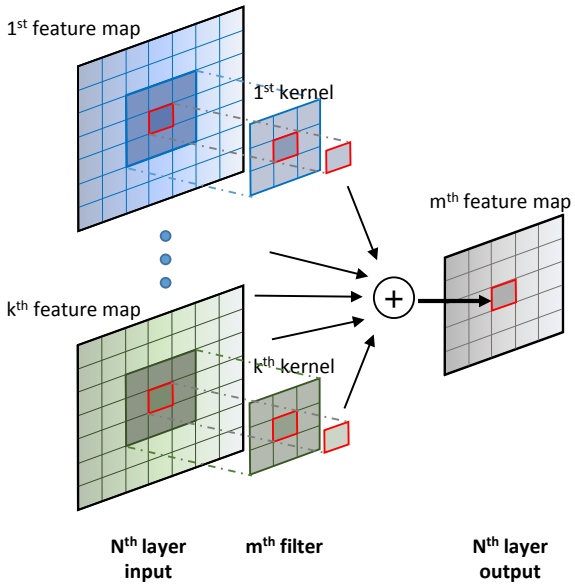


Fig. 2. Demonstration of calculating an output feature at each layer. Each input is convolved with a different kernel and summed together.

the utilization of computing power. The allocator combines data reuse and data concatenation to alleviate the bandwidth bottleneck. This step is essential to exploiting the parallelism of DCNNs, i.e., parallelism over multiple convolutional planes.

1) Overview of Convolution Operation in DCNNs: DCNNs contain hundreds of filters at each layer to extract features by convolving inputs. Convolution with each filter includes multiple 2D convolutions and an accumulation step:

$$F_f(x,y) = \sum_c F_f(c,x,y) \tag{3}$$

where c is the number of input feature maps and $F_f(c,x,y)$ is the output of the 2D convolution as given in Equation 2. This operation is depicted in Figure 2, where each line refers to a 2D convolution operation and the arrows going to the same output are summed together.

The breakdown of this operation in our system is as follows:

1) Each convolution engine from each collection processes a 2D input feature map by convolving it with a 2D filter.
2) The results of the convolution of the first collection are streamed into the neighbor collection's stream adder for summation.
3) The stream adder performs an element-wise summation of the upcoming stream from the neighbor collection and the stream coming from the convolution operator in its own collection.
4) The results of the stream adder go to the neighbor collection's stream adder for further accumulation of 2D convolutions.

There are generally not enough resources to calculate all the 2D convolutions needed to produce one output map. Hence, the outputs of each DMA transaction (partial summation over c in Equation 3) are saved as intermediate values. In the next transaction, the intermediate values are streamed into the custom hardware for further accumulation with the outputs from the other 2D convolution operations.

2) Node Parallelism: The node parallelism, in this context, refers to a parallelization across images within a module. In other words, each 2D convolution across images is an independent operation and can be parallelized. However, each instance requires its own input and output streams. Ideally, the parallelism is equal to the number of collections that can fit into an FPGA. Nevertheless, due to the limited bandwidth of the custom hardware, streaming input and output for each operation requires an optimized routing scheme for the available resources. We combine data reuse and data concatenation to achieve maximum node parallelism.

To justify the need for an optimized routing scheme, we first calculate the utilization of the naive approach for a system with four ports (I/O bus) and eight collections as shown in Figure 3(a). Let's say we use one convolution engine from each collection. Though calculating one output map generally requires hundreds of 2D convolutions, the system can only transfer three input streams, as the forth must be kept off the intermediate values. In such a scenario the maximum utilization reaches 38% of the collections.

3) Input Reuse: To reduce repeated DMA memory access, we want to reuse streamed input values as much as possible. For example, assuming the availability of four ports and eight convolution engines, we send four different input values to the first four convolution engines and transfer the same four input values to the other four convolution engines. As the accelerator processes the same input with different filters, we take advantage of input reuse, as depicted in Figure 3(b). In other words, for each output map we need to calculate Equation 3 (Equation 3 uses different filters for each output map, but uses the same input maps). Because we cannot complete the computations of Equation 3 with one cycle, we calculate the intermediate values for two output maps together. While a related routing scheme was proposed in [39], this method only provided full utilization where streaming of the intermediate values was not required. When intermediates are created and need further processing, two ports would need to be used to stream them in and out, giving only 50% utilization in general.

This work achieves optimum utilization by taking advantage of both the data concatenation and this routing scheme at the same time.

4) Concatenation of Data: The 32-bit AXI bus carries data words represented in Q8.8 format (8 bits for integers and 8 bits for fractions) via DMA transfer. Since each word uses only 16 bits, the 16 most significant bits remain vacant, lowering effective data bandwidth by half (Figure 3(a)). Therefore, two consecutive data words can be concatenated in a single bus during the transfer, which can increase the effective bandwidth. However, if data are not produced together, a waiting time in the processing pipelines occurs due to the delay between transferring and processing.

If two consecutive data words are produced together, and resultantly consumed together, they can be concatenated, increasing the bandwidth without causing any delay. This routing scheme encourages concatenation because two intermediate

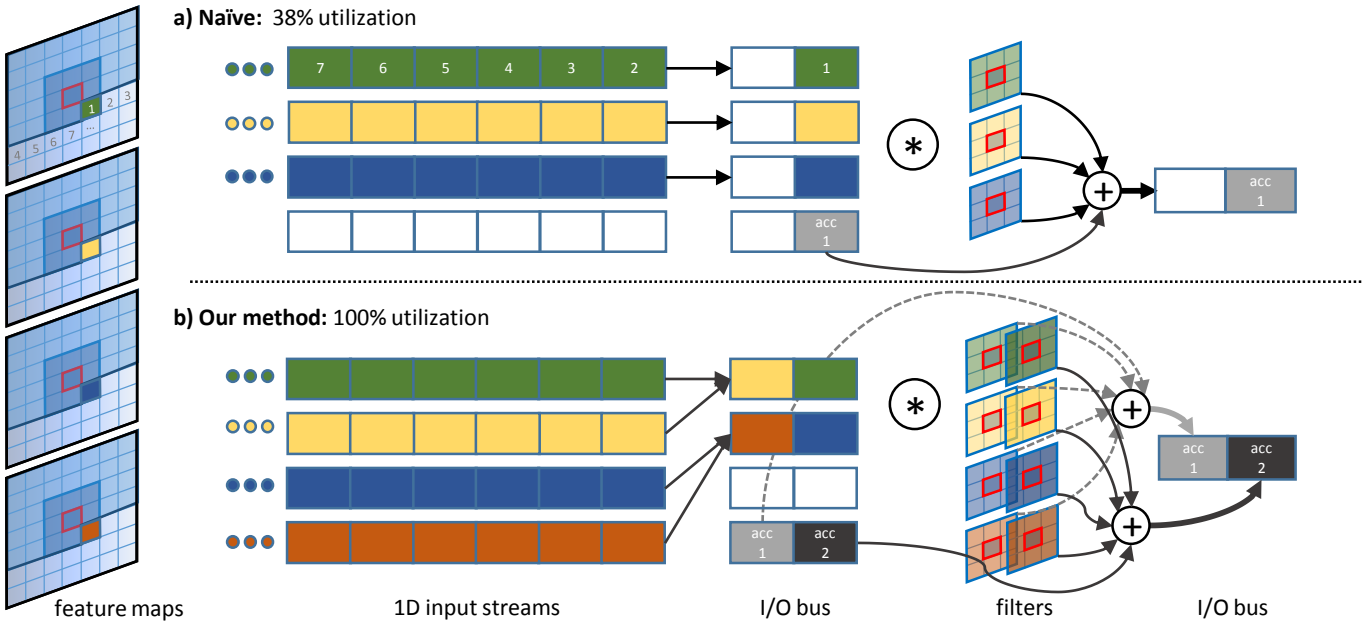


Fig. 3. Routing schemes for 3D convolutions. The naive approach is to process the operations to calculate one output map, as shown in the Figure 3(a). This approach streams 3 inputs and an intermediate value (generally) and reaches a 38% utilization of computational resources. Our approach, 3(b), improves upon the naive approaches and reaches 100% utilization, using data concatenation and input reuse.

values or two output values are produced concurrently. Those values will also be used together later on. With the approach from Figure 3(a), concatenation of outputs would not be possible since each layer requires hundreds of convolution operations. It would therefore not be possible to produce two outputs in one cycle. However, with the approach from Figure 3(b)), outputs and intermediates are created together, so they can be concatenated and brought together in the next cycle in order to calculate the next layers' outputs. Figure 3 is depicted for four ports for simplicity. In our case, we have eight ports and multiple convolution engines, e.g. 64 - 3×3 convolution engines. This routing scheme is flexible for any number of ports and computational power, and provides benefits when the memory access is the limitation. It effectively increases the bandwidth eightfold.

Note that two data words are encapsulated into a 32-bit stream (each word is 16-bit). As soon as the stream reaches the memory router, two data words are outputted and fed to different convolution engines.

5) Flow of Resource Allocator: The flowchart of the system is shown in Figure 4. The table containing rows of instructions from the network parser is scheduled row by row (as long as resources permit). As described in subsection IV-B3, when resources are allocated, each half of the collections is allocated for each set of filters. As long as ports and collections are available, resources continue to be allocated, and once one kind of resource is completely allocated, the transaction is performed. If the calculations have not been completed for an output feature map, the outputs of the transaction are saved as intermediate values. In the next cycle, the system checks if there are intermediate values that need to be processed further. If so, it allocates resources for them as well. This routing scheme is very flexible and can process DCNNs with any number of filters and layers.

C. Hardware Configurator

The hardware configurator is the lowest level of our control method to execute a DCNN application. It provides an environment for communication between the resource allocator and the custom hardware through AXI memory-mapped and high-throughput drivers. For a given routing topology (from the resource allocator), this module produces corresponding operation codes as output. It uses a static routing method so that it only has to be determined once (since codes are generated one time at the start, and the hardware loops over the operation sequence during actual execution).

The hardware configurator produces and serializes operation codes for coalesced memory access. The sequence of codes resides in the contiguous memory, which facilitates DMA transfer and minimizes lead time without scatter-gather addressing. The length of operation codes for a large-scale network is often greater than 30 Mbytes for each frame. The high number of operations must then be directed to the accelerator, and this consumes a large portion of transfer time within the overall execution time. An additional performance boost can be created by caching the operation codes in the on-chip memory within the custom logic, but this comes at the cost of an area. In this case, the hardware accelerator only requires a single code to trigger a certain routing topology, providing us a speed close to the theoretical maximum.

V. Experimental Results

We compare the performance of our custom hardware with other platforms. We present our custom hardware with two different configurations: our novel routing scheme (input

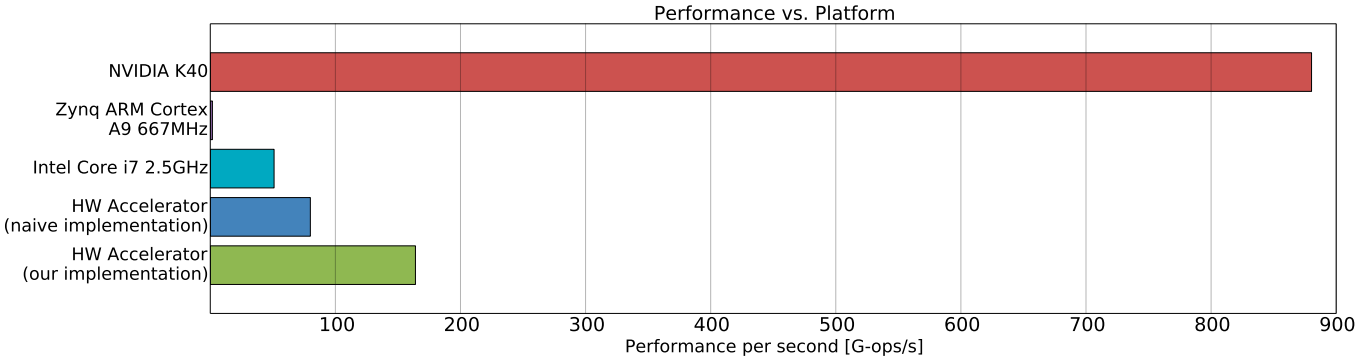


Fig. 5. Performance per second test over one layer network on different platforms. The input is an 500×500 RGB image. The network has 6×6 128 filters, 4×4 max-pooling and non-linearity operation. We also compare the hardware accelerator that is configured with the naive implementation described above and our implementation.

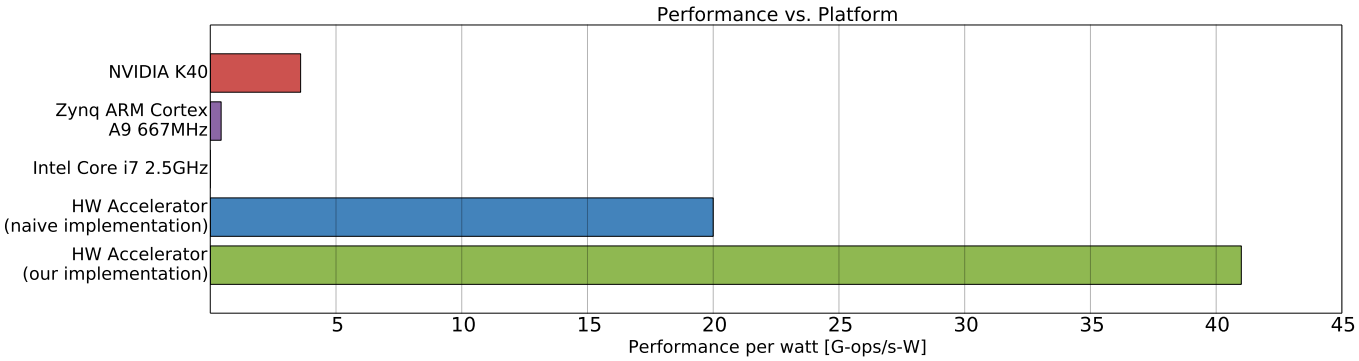


Fig. 6. Performance per watt test over one layer network on different platforms. The input is an 500×500 RGB image. The network has 6×6 128 filters, 4×4 max-pooling and a ReLU non-linearity operation. We also compare the hardware accelerator that is configured with the naive implementation described above and our implementation.

reuse and concatenation) and the naive routing scheme. Other platforms include Intel Core-i7 2.5GHz CPU, NVIDIA K40, and ARM Cortex A9 processor. In our first experiment, we run a single layer network. This experiment uses an input image of $3 \times 500 \times 500$, where 3 is the number of channels (e.g. RGB images). The single layer network consists of 3×128 convolution operations with 6×6 filters, a max-pooling operation with 4×4 window size and a non-linearity operation. This experiment is designed to showcase the peak performances of platforms with large image sizes and filters.

As can be seen in Figure 5, the performance per second of NVIDIA K40 is five times greater than the system presented here. However, desktop GPUs have a TDP consumption around $225 - 300$ watts (245 for the NVIDIA K40), whereas our platform consumes a maximum of 4 watts. Figure 6 presents the comparison performance per watt. In this case, our implementation is 10 times better than the GPU alternative in performance per watt. In the comparison of the novel vs. the naive routing scheme, our system reports 164 G-ops/s, while the naive approach achieves just 80 G-ops/s. The hardware accelerator with the optimized control method is 96 times faster than the baseline embedded processor, a dual-core ARM Cortex A9. The peak performance of the presented system is 247 G-ops/s when the filters are larger (12×12). In the following section, more experimentations with a multi-layer network are utilized for the presented applications.

VI. Applications

In this section, we present two different applications that we run on our system: object classification and object detection. First, the details of the architecture and training are presented, and then the applications are explained, providing screen shots for each application. Videos of the applications are available online[1].

A. Network Architecture and Training

The network set-up has five layers of convolution, max-pooling, ReLU modules, and three fully connected linear layers. The parameters of the network architecture are given in Table I. In the first layer, inputs are padded with zeroes to obtain the information in the borders of the images. Padding is applied to the fourth and fifth layers as well. In the first layer, max-pooling is applied by a stride of four. The input size of the network is $3 \times 231 \times 231$ and the spatial size of the input of each layer decreases as we move to the end of the network, as given in Table I. The output of the network is fed to a soft-max operation which normalizes the summation of values to one so they represent a probability distribution.

[1]http://web.ics.purdue.edu/~adundar/

Layer	1	2	3	4	5	6	7	8
Stage	conv+max	conv+max	conv+max	conv+max	conv	full	full	full
#channels	48	64	64	64	32	128	128	20
Filter size	11×11	5×5	3×3	3×3	3×3	-	-	-
Pooling size	4×4	2×2	2×2	2×2	-	-	-	-
Pooling stride	4×4	2×2	2×2	2×2	-	-	-	-
Zero-Padding size	5×5	-	-	1×1	1×1	-	-	-
Spatial input size	231×231	58×58	27×27	12×12	6×6	6×6	1×1	1×1

TABLE I
THE SPECIFICATIONS OF THE NETWORK USED FOR THE APPLICATIONS. INPUT SIZE OF EACH LAYER IS CALCULATED BASED ON THE ZERO-PADDING SIZE, PARAMETERS OF CONVOLUTION AND POOLING OPERATIONS.

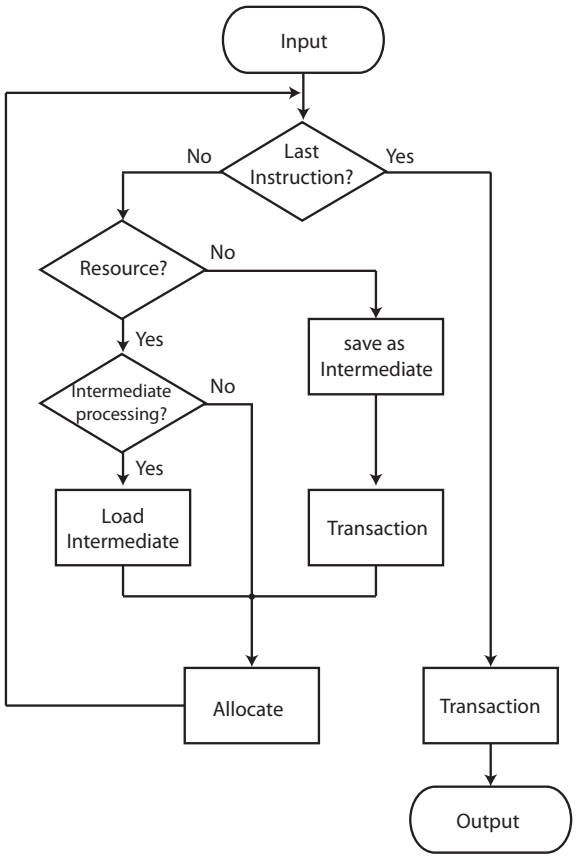


Fig. 4. The flowchart of the control method. Each instruction is scheduled as long as resources are available. Otherwise the instructions that are scheduled are processed. If there are not enough resources to complete the calculations for one output, the partial output from the calculations is saved as an intermediate value which comes in the later transaction for further processing.

This network is trained with twenty categories, which is the size of the network output. The network is trained in the Torch environment [40] on NVIDIA K40.

DropOut [8] with a rate of 0.5 is employed on the fully connected layers (sixth and seventh) in the classifier. Dropout sets the output of each hidden layer to zero with a probability of 0.5. In this process, only half of the network is trained in each forward-backward pass. As a result, complex co-adaptations of neurons decrease, as does over-fitting. In the test period, outputs of neurons are multiplied by 0.5, so the whole network contributes to the prediction.

We follow the input preparation method as [4]. It starts by cropping images that are 25×25 larger than the network's input size, 263×263. Each image is down-sampled so that the smallest side of the image is 263. Images are cropped from the center to obtain 263×263 size images. To avoid over-fitting and increase the training dataset, 231×231 random patches are extracted from the images at each training step; some patches are randomly flipped horizontally. Finally, as pre-processing, patches are normalized by subtracting the mean and dividing by the standard deviation of the dataset. The same network is used for both applications.

B. Object Classification

The first task is using the network to classify objects (i.e., assign trained classes to an image). Figure 7 shows a screenshot from a video we recorded. Because an image can have multiple objects, we display the top 2 probabilities. Images from the camera were down-sampled to the size of 231×231 and fed to the network. The output of the network is the probability distribution of each class in the image. As demonstrated in the figure, the system is able to recognize two presented objects correctly, despite the fact that they are not centered.

Note that when just one object is present, the second highest probability can be low or irrelevant, interpreted as noise.

In Table II, a comparison of the performance per watt of our system for this application is presented. There is a drop in the performance in comparison to Figure 6 because of the small filter sizes that decrease the weight level parallelization. However, our system still outperforms all other presented platforms in performance per watt. In Table III, the performance numbers for each layer in the classification application are given. The number of operations needed for each layer and the time it takes to process each layer in our system are provided in this table. Differing numbers of filters, filter sizes, or input sizes to each layer make a big difference in the number of computations and the overall performance. Note that only the convolutional layers are processed in each platform because a linear layer is not implemented in our system.

C. Object Detection

Object detection is a more complex task than simple classification since a bounding box for the predicted object must be

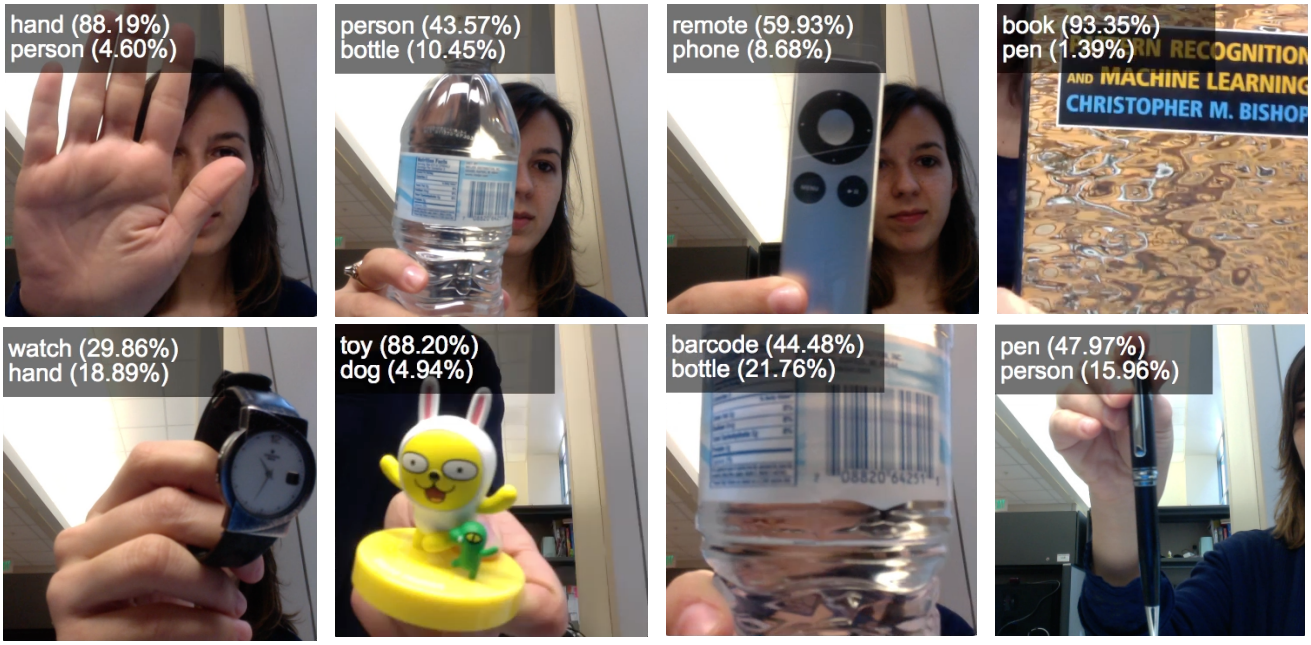


Fig. 7. Object classification: DCNN takes each frame and produces a probability distribution among the classes the network is trained with. The top 2 classes with the highest probabilities are displayed on the left corner of images with the probabilities. The network is able to recognize two objects at the same time. When there is only one object in the image and the second probability is low, it can be considered noise.

	Object Classification		Object Detection	
Performance	Per Second G-ops/s	Per Second - Watt G-ops/s-W	Per Second G-ops/s	Per Second - Watt G-ops/s-W
NVIDIA K40	573.3	2.34	828.7	3.38
Zynq ARM Cortex A9 667MHz	1.7	0.43	1.7	0.43
Intel Core i7 2.5GHz	88.6	1.1	78	1.0
HW Accelerator (Naive implementation)	58.2	14.6	81.2	20.3
HW Accelerator (Our implementation)	115.2	28.3	179	43.5

TABLE II
PERFORMANCE PER SECOND AND PERFORMANCE PER SECOND-WATT NUMBERS OVER CLASSIFICATION AND DETECTION APPLICATIONS ON DIFFERENT PLATFORMS.

HW Accelerator (Our implementation)			
Network given in Table 1		Performance	
	G-ops	ms	G-ops/s
layer 1	1.86470	15.28	122.0
layer 2	0.44827	4.27	105.0
layer 3	0.04616	0.67	69.0
layer 4	0.00739	0.23	32.1
layer 5	0.00059	0.10	5.9
Overall	2.36711	20.55	115.2

TABLE III
PERFORMANCE PER SECOND AND PERFORMANCE PER SECOND-WATT NUMBERS FOR EACH LAYER FOR CLASSIFICATION APPLICATION WITH OUR IMPLEMENTATION.

returned. That is, the network should predict both the correct category and the location of the object. Also there might be multiple objects in a scene, and these objects might not belong to any trained category.

To return the bounding box, we follow the *sliding-window* approach without any pre-processing (in a similar manner as [41]–[43]). In DCNNs, this approach is efficient since they share computations that are common to the overlapping regions. The output of the network is a three-dimensional feature map, where one dimension has the information of the probability distributions and the other two dimensions have the information of the spatial coordinates.

To remove closely detected duplicates among the detections that are spatially very close, the one with the highest probability is chosen while the rest are discarded. Because the network is not trained with all the visible categories in each scene, a fixed low threshold of probability is set for a prediction to be valid. This is necessary because the network consistently attempts to pick a category for each patch despite the possibility that there is not a recognizable object.

In Figure 8, the detection application's screenshots of a

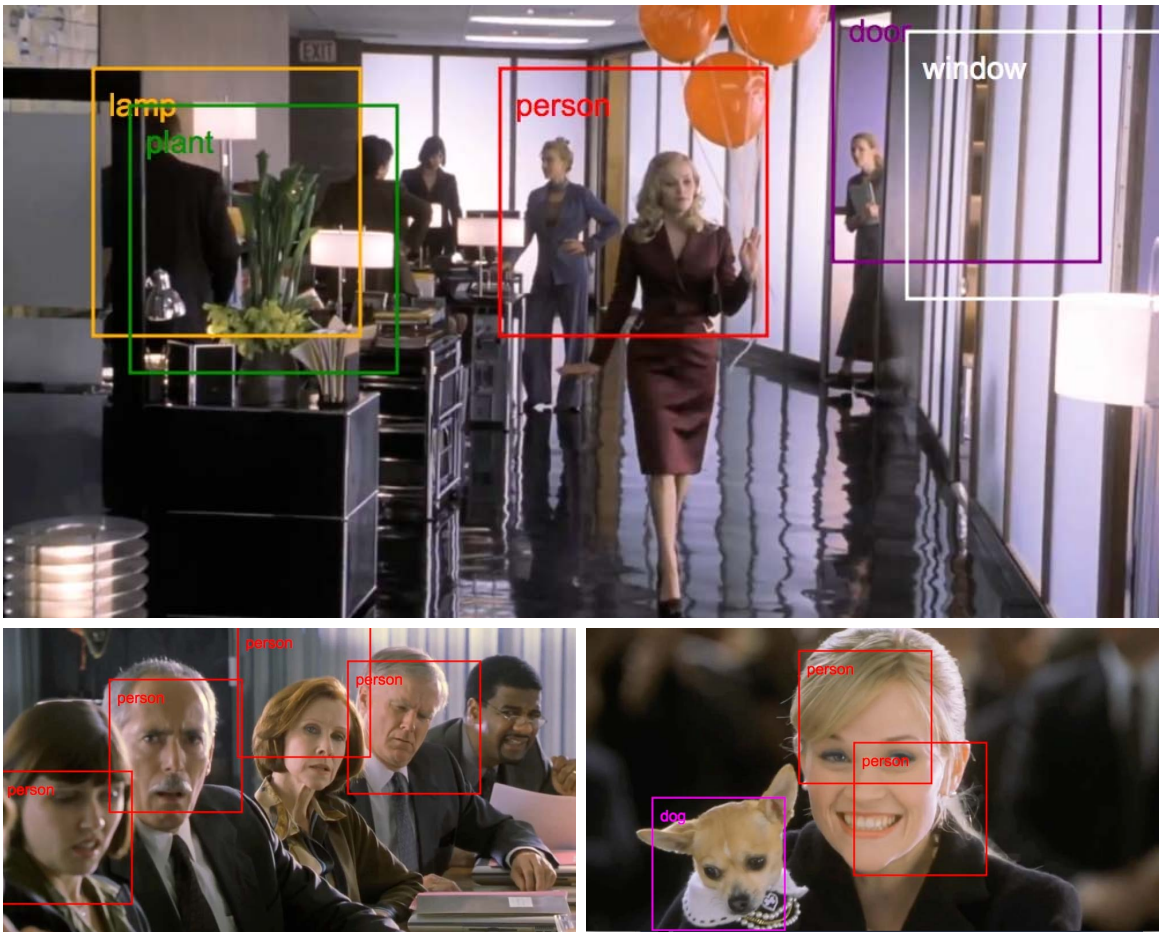


Fig. 8. Object detection: DCNN scans the frame in a sliding window fashion. A simple post-processing is applied to show DCNNs capabilities. Detections above a threshold are displayed after pruning the detections in close regions. Detections can be improved with more sophisticated post-processing or by using multiple scales.

movie trailer are presented. The video contains 360×720 RGB images. Frames of the video are fed to the network after subtracting the mean and dividing by the standard deviation of the training dataset. This detection process could be further improved by implementing more categories (so that the system is able to detect more objects), using the multi-scale approach, or combining the prediction of DCNNs with pre or post-processing.

In Table II, comparisons of the performance per second and per watt of our system are presented for this application. The performance numbers for our system improve compared to the classification application because of the larger image sizes and the streaming architecture of our system.

VII. Conclusion

We present a novel control method for a hardware-accelerated real-time implementation of deep convolutional neural networks (DCNN) on an embedded platform. This control method achieves full utilization of the resources of the hardware by exploiting the characteristics, architecture, and configuration of DCNNs. Our method combines data concatenation and data reuse for the maximum speed-up.

Our results show that our system is extremely performance efficient. We also test our system with two different applications: object classification and object detection. The applications using DCNNs are extensive, including tracking [44], [45], action recognition [46], face recognition [47], pixel-level scene labeling [48] and stereo matching [49].

The proposed routing scheme is implemented using custom hardware with eight ports and four collections; each has multiple convolution operations, but this routing scheme can be uncovered in any other configuration. While specialization emerges for energy efficiency, flexibility of the system is also important. The system presented here can run different architectures of DCNNs, with any numbers of filters and layers, using the implemented control method. Furthermore, this system can be used for generic image processing applications which use convolution-like data flow. For example, convolution operations can be replaced with sum-of-absolute-differences (SAD) or sum-of-square-differences (SSD) operations, widely used in tracking and motion estimation algorithms. Additional examples demonstrating the usability of this system include SIFT, median-filtering, and video processing, among others.

ACKNOWLEDGMENT

We would like to thank anonymous reviewers for their helpful comments. Work was supported by the Office of Naval Research (ONR) grants 14PR02106-01 P00004 and MURI N000141010278.

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. Image 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] D. Grangier, L. Bottou, and R. Collobert, "Deep convolutional networks for scene parsing." Citeseer.

[6] 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.

[7] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional neural networks," *arXiv preprint arXiv:1311.2901*, 2013.

[8] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.

[9] M. A. Erdogdu and A. Montanari, "Convergence rates of sub-sampled newton methods," in *Advances in Neural Information Processing Systems*, 2015, pp. 3034–3042.

[10] M. A. Erdogdu, "Newton-stein method: A second order method for glms via stein's lemma," in *Advances in Neural Information Processing Systems 28*, 2015, pp. 1216–1224.

[11] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2010, pp. 257–260.

[12] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[13] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," vol. abs/1410.0759, 2014.

[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*. IEEE, 2009, pp. 878–885.

[15] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. Graf, "A massively parallel coprocessor for convolutional neural networks," in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, 2009, pp. 53–60.

[16] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 273–284.

[17] 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.

[18] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 24–35.

[19] C. Yan, Y. Zhang, J. Xu, F. Dai, J. Zhang, Q. Dai, and F. Wu, "Efficient parallel framework for hevc motion estimation on many-core processors," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 24, no. 12, pp. 2077–2089, 2014.

[20] C. Yan, Y. Zhang, J. Xu, F. Dai, L. Li, Q. Dai, and F. Wu, "A highly parallel framework for hevc coding unit partitioning tree decision on many-core processors," *Signal Processing Letters, IEEE*, vol. 21, no. 5, pp. 573–576, 2014.

[21] A. R. Omondi and J. C. Rajapakse, *FPGA implementations of neural networks*. Springer, 2006.

[22] P. Sermanet, K. Kavukcuoglu, and Y. LeCun, "Eblearn: Open-source energy-based learning in c++," in *Tools with Artificial Intelligence, 2009. ICTAI'09. 21st International Conference on*. IEEE, 2009, pp. 693–697.

[23] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Implementing neural networks efficiently," in *Neural Networks: Tricks of the Trade*. Springer, 2012, pp. 537–557.

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

[25] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.

[26] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Two*. AAAI Press, 2011, pp. 1237–1242.

[27] F. Nasse, C. Thurau, and G. A. Fink, "Face detection using gpu-based convolutional neural networks," in *Computer Analysis of Images and Patterns*. Springer, 2009, pp. 83–90.

[28] D. Scherer, H. Schulz, and S. Behnke, "Accelerating large-scale convolutional neural networks with parallel graphics multiprocessors," in *Artificial Neural Networks–ICANN 2010*. Springer, 2010, pp. 82–91.

[29] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *Proceedings of The 30th International Conference on Machine Learning*, 2013, pp. 1337–1345.

[30] 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, pp. 109–116.

[31] 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.

[32] 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.

[33] M. Peemen, A. 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*. IEEE, 2013, pp. 13–19.

[34] P. Merolla, J. Arthur, R. Alvarez-Icaza, A. Cassidy, J. Sawada, F. Akopyan, B. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. Esser, R. Appuswamy, B. Taba, A. Amir, M. Flickner, W. Risk, R. Manohar, and D. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, pp. 668–673, August 2014.

[35] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, 2015.

[36] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[37] J. Jin, V. Gokhale, A. Dundar, B. Krishnamurthy, B. Martini, and E. Culurciello, "An efficient implementation of deep convolutional neural networks on a mobile coprocessor," in *Circuits and Systems (MWSCAS), Proceedings of 2014 IEEE 57th International Midwest Symposium on*, 2014.

[38] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 682–687.

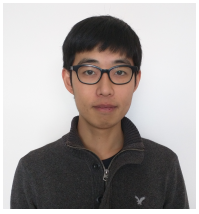
[39] A. Dundar, J. Jin, V. Gokhale, B. Martini, and E. Culurciello, "Memory access optimized routing scheme for deep networks on a mobile coprocessor," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

[40] R. Collobert, C. Farabet, and K. Kavukcuoglu, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[41] M. Delakis and C. Garcia, "text detection with convolutional neural networks." in *VISAPP (2)*, 2008, pp. 290–294.
[42] C. Garcia and M. Delakis, "A neural architecture for fast and robust face detection," in *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, vol. 2. IEEE, 2002, pp. 44–47.
[43] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *arXiv preprint arXiv:1312.6229*, 2013.
[44] J. Fan, W. Xu, Y. Wu, and Y. Gong, "Human tracking using convolutional neural networks," *Neural Networks, IEEE Transactions on*, vol. 21, no. 10, pp. 1610–1623, 2010.
[45] 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.
[46] S. Ji, W. Xu, M. Yang, and K. Yu, "3d convolutional neural networks for human action recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 1, pp. 221–231, 2013.
[47] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*. IEEE, 2014, pp. 1701–1708.
[48] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 8, pp. 1915–1929, 2013.
[49] J. Žbontar and Y. LeCun, "Computing the stereo matching cost with a convolutional neural network," *arXiv preprint arXiv:1409.4326*, 2014.

Aysegul Dundar received a Bs.C. degree with honors in Electrical and Electronic Engineering from Bogazici University, Istanbul, Turkey in 2011. She is pursuing a Ph.D. at the Weldon School of Biomedical Engineering at Purdue University under the direction of Eugenio Culurciello.

Her research is focused on synthetic models of the human vision system, specifically deep convolutional neural networks in hardware. Her work on nn-X embedded vision systems has been featured on the MIT Technology Review and involved in the start-up company Teradeep.



Jonghoon Jin received a B.E. degree with honors in Electrical Engineering from Korea University, Seoul, Korea, in 2011. He continued his studies in Electrical and Computer Engineering at Purdue University where he joined the artificial vision research laboratory. He received a M.S. in Electrical and Computer Engineering from Purdue University in 2014 and is currently pursuing a Ph.D. degree.

His research is focused on hardware software co-design for deep learning algorithms. He is a main developer of nn-X embedded vision systems and his work is involved in the start-up company Teradeep.

Berin Martini received a B.Sc. degree with Honors in Physics from University of Melbourne, Australia, in 2001.

He worked for the Defence Science and Technology Organisation (DSTO) for a number of years until 2007, when he began working at eLab at Yale University, New Haven, CT, followed by Purdue University, West Lafayette, IN, as a Research Associate.

Eugenio Culurciello (S'97-M'99) received a Ph.D. degree in Electrical and Computer Engineering in 2004 from Johns Hopkins University, Baltimore, MD.

He is an Associate Professor of the Weldon School of Biomedical Engineering and an Associate Professor of Psychological Sciences in the College of Health & Human Sciences at Purdue University, West Lafayette, IN, where he directs the e-Lab laboratory. He is the author of Silicon-on-Sapphire Circuits and Systems, Sensor and Biosensor Interfaces (McGraw-Hill, 2009). His research interest is in analog and mixed-mode integrated circuits for biomedical instrumentation, synthetic vision, bio-inspired sensory systems and networks, biological sensors, and silicon-on-insulator design.

Dr. Culurciello is the recipient of the Presidential Early Career Award for Scientists and Engineers (PECASE) and Young Investigator Program from ONR, the Distinguished Lecturer of the IEEE (CASS).