A Dynamically Configurable Coprocessor for Convolutional Neural Networks

Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula and Srihari Cadambi
NEC Laboratories America, Inc.
4 Independence Way, Princeton NJ 08540.
{chak, murugs, jakkula, cadambi}@nec-labs.com

ABSTRACT

Convolutional neural networks (CNN) applications range from recognition and reasoning (such as handwriting recognition, facial expression recognition and video surveillance) to intelligent text applications such as semantic text analysis and natural language processing applications. Two key observations drive the design of a new architecture for CNN. First, CNN workloads exhibit a *widely varying mix of three types of parallelism*: parallelism within a convolution operation, intra-output parallelism where multiple input sources (features) are combined to create a single output, and inter-output parallelism where multiple, independent outputs (features) are computed simultaneously. Workloads differ significantly across different CNN applications, and across different layers of a CNN. Second, the number of processing elements in an architecture continues to scale (as per Moore's law) much faster than off-chip memory bandwidth (or pin-count) of chips. Based on these two observations, we show that for a given number of processing elements and off-chip memory bandwidth, a new CNN hardware architecture that dynamically configures the hardware on-the-fly to match the specific mix of parallelism in a given workload gives the best throughput performance. Our CNN compiler automatically translates high abstraction network specification into a parallel microprogram (a sequence of low-level VLIW instructions) that is mapped, scheduled and executed by the coprocessor. Compared to a 2.3 GHz quad-core, dual socket Intel Xeon, 1.35 GHz C870 GPU, and a 200 MHz FPGA implementation, our 120 MHz dynamically configurable architecture is 4x to 8x faster. This is the *first CNN architecture to achieve real-time video stream processing* (25 to 30 frames per second) on a wide range of object detection and recognition tasks.

Categories and Subject Descriptors

C.1.3 [Other Architecture Styles]: Adaptable architectures, Neural nets, pipeline processors.

General Terms

Design, Experimentation, Performance.

Keywords

Convolutional Neural Networks, Dynamic Reconfiguration, Parallel Computer Architecture.

ISCA'10, June 19–23, 2010, Saint-Malo, France.

1. INTRODUCTION

Feed-forward multilayer neural networks [12] are computational models that are widely used in diverse domains such as video and image processing [22], medical diagnosis systems [14] and financial forecasting [15]. These computation models serve one of two roles: pattern recognition to provide a meaningful categorization of input patterns, or functional approximation where the models find a smooth function that approximates the actual mapping between input and output patterns. A vast majority of these computational models are still implemented in software on general-purpose and embedded processors. However, these processors do not fully exploit the parallelism inherent in these computational models. Numerous custom hardware implementations have also been proposed to parlay the abundant parallelism inherent in these computational models into significantly higher performance [21].

Traditional pattern recognition systems use two distinct steps to recognize individual patterns. First, a feature extractor transforms the input patterns into short strings of symbols (low-dimensional vectors) that can be easily matched or compared. These features are relatively invariant with respect to transformations and distortions that do not change the inherent nature of the patterns. The feature extractor is usually rather specific to the task. It is also the focus of most design effort, because it is entirely hand-crafted. Second, a classifier categorizes the feature vectors into a few classes. The classifier is usually general-purpose, and a standard, fully-connected multi-layer neural network can be automatically trained to do the classification. However, recognition accuracy is largely determined by the ability of the designer to extract an appropriate set of features. Feed-forward, multi-layer artificial neural networks like the Convolutional Neural Networks (CNN) [2][11][13][16][17][20][23] have found increasing use in several new applications because they have the potential to process vast amounts of labeled data to *automatically* learn and extract complex features. CNNs are especially attractive because they can recognize visual patterns directly from pixel images with very minimal preprocessing of the image. CNNs can easily recognize patterns with extreme variability (such as handwritten characters). Also, their recognition ability is not impaired by distortions or simple geometric transformations (for example, translation or rotation) of the image.

Neural networks have multiple layers of neurons (an input layer, an output layer and one or more hidden layers). Every connection between an input and a neuron is assigned a value called *weight*. Each neuron computes a weighted sum of all its inputs, followed by a non-linear or sigmoid function to restrict its output value within a reasonable range. Neurons in the hidden layers are also called the *hidden* units. As an example, consider one layer of a multi-layer neural network and the task of

connecting 1024 inputs to 784 hidden units to produce a set of 784 output values. The 1024 inputs can be the 1024 pixel values in a 32 x 32 image. The 784 hidden units produce 784 output values (also called as a *feature map*) which can be interpreted as pixels in a 28 x 28 output image. A fully-connected neural network connects every input to every hidden unit, resulting in a network that has 1024*784 = 802,816 different weights (free parameters). Learning these weights will require a very large number of training instances. Furthermore, a fully-connected architecture ignores the topology of the input image because input values can be presented in any order without affecting the outcome of the training. On the contrary, images have strong two-dimensional local structure where pixels that are spatially nearby are highly correlated. Therefore, each hidden unit in a CNN is connected to only on a small number of inputs, say 25 inputs. These 25 inputs, also called as the *receptive field* of the hidden unit, correspond to a 5 x 5 pixel area in the input image. Only 25 weights are now necessary to connect the inputs to a hidden unit. Since such an elementary feature detector is likely to be useful across other 5 x 5 pixel areas in the input image, many different hidden units are used to cover the entire image. Receptive fields of these hidden units can overlap, and the degree of overlap can be pre-specified. Hidden units, whose receptive fields are located at different places on the image, can be forced to detect the same elementary feature by reusing the same 25 weights. Hidden units that share the same set of weights form a *plane*, and outputs of units in a plane constitute the feature map. Hidden units in a plane can together extract visual features like edges, corners etc. If we represent the 25 distinct weights as a 5 x 5 matrix (also called as the *kernel* matrix), then outputs of all the 784 hidden units can be computed as the convolution of the 32 x 32 input image with the 5 x 5 kernel matrix [1].

After the convolution step, we have a 28 x 28 image in which a feature of interest has been detected, and the exact location of the feature becomes less important. The value of every hidden unit is also subjected to a squashing function (non-linearity). A simple way to reduce the precision with which the distinctive features are encoded in the image is to reduce the spatial resolution of the image by using sub-sampling. This also reduces the sensitivity of the outputs to shifts or distortions. A typical CNN has multiple layers of hidden units (with multiple feature maps per layer) to perform complex object recognition tasks.

Evaluating a trained CNN involves performing a huge number of convolutions with considerable data movement. Convolution computation is a performance bottleneck, but reducing overheads of data movement is also necessary to accelerate the performance of a CNN. Consider a simple object (face) recognition application that is used on relatively high resolution streaming images. With a 320x240 (QVGA) image, a CNN network that can be used to identify faces within all possible 32x32 windows in the image runs at approximately 6.5 frames per second on a 2.5GHz Intel Xeon processor when optimized using BLAS (Intel MKL v11). Multi-threading this to 4 and 8 cores on quad-core and dual quad-core machines only improves the speed by a little over 2x due to synchronization overheads, and the fact that different threads share common inputs. Therefore, the most optimized software implementation on state-of-the-art processors struggles to achieve about 8 to 10 frames per second when we analyze VGA (640x480 pixels) images. VGA (or larger) images are more realistic in practical use-case scenarios such as security cameras. Can the abundant parallelism in CNN workloads be leveraged by custom architectures to improve the feed-forward processing speeds to be close to real-time (25 to 30 frames per second)? To answer this question, we investigate a dynamically reconfigurable architecture in which the hardware parallelism can be tailored to suit the parallelism offered by the specific application workload.

1.1 Related Work

Implementation of the convolution operation in hardware has been studied extensively. Both FPGA and LSI implementations have been proposed. Digital media processors with a large number of high-speed multiply-and-accumulate (MAC) units have been used to implement the convolution operation [18]. LSI architectures using a mixed analog-digital approach [20] as well as several FPGA-based implementations have also been proposed [3][4][5][6][7][19][24].

Fast implementations of a 2D convolution core are necessary but not sufficient to accelerate the CNN workload. A host processor can carve out the convolution operations from the CNN workload and off-load the convolutions to the hardware implementation. For each off-load, the host processor provides the image and weight values, and retrieves the convolved output image from the accelerator. Some optimizations are possible to store kernel values or some intermediate data on the accelerator. However, for CNNs used in practice, the data dependencies within a layer (sub-sampling and non-linearity operations) and across multiple network layers of the CNN, the management of significant amount of intermediate data over a slow host to hardware accelerator link, and the detailed orchestration of complex control and data flows by the host processor quickly offset performance gains obtained by using a fast hardware core only for 2D convolutions. There are no reported LSI implementations of CNNs but several software implementations on GPUs exist [8][25]. However, none of the prior implementations are able to process video feeds (640 x 480 pixel VGA frame) in *real-time* (about 25 to 30 frames per second).

An FPGA implementation of CNN was reported recently [9]. This architecture uses one hardware convolver for data processing, and a general-purpose soft-processor for control, all implemented on a Virtex 4 FPGA from Xilinx. The self-contained unit was developed for video processing in mobile robot applications. In contrast, we use a very different approach. We employ a configurable bank of hardware convolvers, a hardwired controller (to manage complex data movement patterns), and configurability hardware that dynamically configures HW resources on-the-fly to match the precise type and mix of parallelism in the CNN workload. In [9], no attempt is made to alter the hardware computing paths to match type or extent of parallelism in the workload.

1.2 Our Contribution

While several earlier efforts have implemented convolutions and neural networks in hardware, to our knowledge, this is the first effort to create a co-processor architecture that automatically analyzes workloads and dynamically configures its hardware and software components to match the exact mix of different types of parallelism in the workload. Our most novel contribution is dynamic configurability and a method to quickly match HW (on-the-fly) to workload characteristics. For the first time, we achieve a 'tipping point' (sustained real-time recognition). We enable new, real-time on-line classification applications that were not possible before.

A high abstraction-level software API, along with a run-time software component allows domain-experts to easily specify and execute arbitrary convolutional neural network workloads. Unlike prior work, our co-processor architecture is forward-scalable. As we scale the number of processing elements, and the available off-chip memory bandwidth, CNN applications continue to automatically improve in performance. Domain-experts do not have to re-write the application. Instead, a software run-time component determines the optimal configuration of processing elements (and memory architecture) for each layer of the CNN workload, and the coprocessor architecture is dynamically configured to match the workload characteristics. This allows the throughput and performance of the coprocessor architecture to be very close to the peak throughput of the individual processing elements. Dynamic configurability in hardware is fast (single-cycle instruction), and it is under the control of software. Our coprocessor (with different functional units) can easily implement different feed-forward neural networks and classifiers such as HMAX [17][27], DBN [29][30]and HoG methods [28].

The rest of this document is organized as follows. In Section 2, we provide a background for CNNs. In Section 3, we discuss the parallelism in CNN workloads. In Section 4, we present a motivating example that illustrates the benefits of a dynamically reconfigurable architecture. In Section 5, we describe the coprocessor architecture and Section 6 describes dynamic configurability. We present architectural evaluation results in Section 7, and conclude in Section 8.

2. CNN: A COMPUTE PERSPECTIVE

We briefly review the forward propagation phase of a CNN. In this paper, we do not consider the learning phase of the CNN that determines the kernel values that will be used in each layer. Rather, we assume that a trained CNN is available and focus on forward propagation. Tasks performed by end-users (to classify images, for instance) involve forward propagation on trained CNNs. There are often stringent real-time performance and power constraints and hardware acceleration is necessary to achieve these goals. Forward propagation is also a core computation in the back-propagation based learning algorithm, and our solution can also accelerate the learning phase.

We provide a computational perspective of the forward propagation phase of a CNN. Figure 1 shows one layer of a typical CNN. Several such layers are cascaded to create a feed-forward neural network where each layer (except the last layer) feeds only the next layer and receives inputs only from the immediately preceding layer. Every CNN layer is a cascade of two distinct computations or sub-layers: convolution, and sub-sampling. Hardware implementations must also consider issues of numerical precision and dynamic range of values computed by the network.

2.1 Convolution Sub-layer

A convolution sub-layer accepts n images $Y_1 \ldots Y_n$ as inputs and produces m intermediate outputs $O_1 \ldots O_m$. To produce the intermediate output image O_i, the images $Y_1 \ldots Y_n$ are first individually convolved with kernels K_{1i} ... K_{ni}. Then, the individual convolution results from each input image are summed, or *aggregated*. A "bias" value is added to each pixel in the aggregated output, and a suitable non-linear function (for example, *tanh*) is used to limit the pixel value to be within a reasonable range. The intermediate output image O_i is roughly the same size as the input images. All kernels used in the convolution

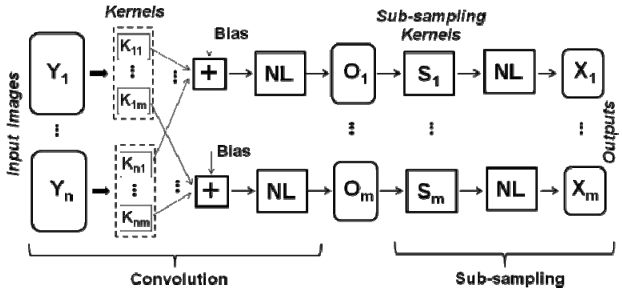

Figure 1: Typical structure of one layer of a CNN.

sub-layer of the CNN are of the same size (rows and columns). However, kernel sizes and the number of input and output images can vary from one CNN layer to another. Mathematically, output image O_i in a convolution sub-layer is as follows:

$$O_i = \tanh\left(bias + \sum_{j=1}^{n} Y_j * K_{ji}\right)$$

Here, $Y_j * K_{ji}$ represents the convolution operation between image Y_j and kernel K_{ji}, and tanh is the non-linear function. Since $m*n$ image-kernel convolutions are performed per CNN layer, from a computational point of view, these convolutions are the most compute intensive portion of a CNN. If the kernels are of size 1, then the CNN degenerates into a regular neural network [1].

For convolving an image with I_r rows and I_c columns with a kernel with W_r rows and W_c columns, the computation workload is $(I_r - W_r) * (I_c - W_c) * W_r * W_c$ multiply-accumulates (MACs). For a CNN layer, we perform $n * m$ convolutions. For each convolution, it is also easy to see that the computation ratio to memory IO ratio is $W_r * W_c$ since for every pixel fetched from memory we perform $W_r * W_c$ MACs. Therefore, CNNs are compute-intensive, especially as the kernel sizes get larger.

2.2 Sub-sampling and non-linearity

In Figure 1, the convolution sub-layer output O_i is sub-sampled. In the simplest case, sub-sampling averages four neighboring elements in O_i to produce a single element in the output image X_i. The output image X_i will have approximately half the number of rows and columns as compared with O_i. In general, O_i can be sub-sampled by using a suitable sub-sampling kernel S_i. After sub-sampling, each element in the output can be subjected to a non-linear operation to produce one pixel of the output X_i. A sub-sampled output image X_i is given by: $X_i = \tanh(O_i \,.\, S_i)$, where tanh is the non-linear function, and the operation $(O_i \,.\, S_i)$ represents sub-sampling of image O_i according to kernel S_i. Compared to the convolution sub-layer, sub-sampling is less compute intensive since it requires, on average, only one MAC operation per input pixel.

3. PARALLELISM IN CNN WORKLOADS

The CNN forward propagation task can be parallelized in several ways. In a multi-layer CNN, due to the feed-forward nature of forward propagation, the data dependencies between successive layers preclude parallel execution of all layers of the CNN. Therefore, task-parallelism across layers is limited and it is also more difficult to exploit.

Operator-level (fine-grained) parallelism: Consider one image-kernel convolution that convolves an image with *Ir* rows and *Ic* columns and a kernel with *Wr* rows and *Wc* columns. Each

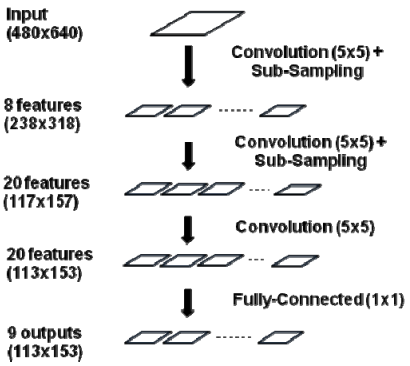

Figure 2: CNN for face recognition (with angle/pose detection).

output pixel requires *Wr*Wc* multiply-accumulations all of which can be performed in parallel. The output pixels themselves are all independent. Numerous 2D-convolver designs exist that exploit the parallelism inherent in the image-kernel convolution, and we employ a standard systolic design. Practical considerations like available memory bandwidth, hardware computation resources, and power considerations do limit the amount of fine-grained parallelism we can exploit in hardware. For example, it is not practical to simultaneously perform all the sub-convolutions in parallel due to the excessive memory bandwidth that will be necessary to bring in almost the entire image in one cycle.

Coarse-grain parallelism: If a CNN layer has *n* input images and *m* outputs, then all *m*n* image-kernel convolutions can, in theory, be performed in parallel. However, for typical *m* and *n*, provisioning enough memory bandwidth to keep *m*n* convolvers busy is impractical. With a smaller number of convolvers, we can extract parallelism in two ways: inter-output and intra-output. We can parallelize the computation of a single output image since it is the sum of *n* input-kernel convolutions. We refer to this as *intra-output parallelism*. Also, multiple output images can be computed in parallel, and we refer to this as *inter-output parallelism*. The key observation is that different layers in a CNN network exhibit vastly different amount of intra-output and inter-output parallelism. As explained in Section 4, due to this variability, fixed hardware architectures have worse performance than an adaptive, configurable architecture.

4. A MOTIVATING EXAMPLE

In this section, we motivate the case for a dynamically configurable CNN coprocessor. We show that CNN workload characteristics change dramatically not only from one application to another, but also across different layers of a single CNN. We designed several hardware architectures and observed the performance of a variety of CNN workloads on each of these architectures. Specifically, we designed a hardware architecture expressly tailored to accelerate workload in a particular layer of the CNN, and evaluated the performance of workloads in other layers of the CNN (and even other applications) on this hardware architecture. We observed that for any given hardware architecture, the CNN layer workload for which the hardware architecture was designed for does exhibit good performance, but a majority of other CNN layer workloads exhibit poor performance. Our study attributes the poor performance of most CNN layer workloads to a mismatch of the computing architecture and the workload characteristics of various CNN layers.

Figure 2 shows a CNN network we use in video applications to detect faces (and their angle and pose). The resolution of the input image is VGA (640x480). The network has 4 layers. The first two layers employ both convolutions as well as sub-sampling (the figure leaves out non-linearity and bias for simplicity). The third layer is only a convolutional layer while the last is a traditional fully-connected layer where all inputs are connected to all outputs. The convolutional kernels (2D array of weights) are of size 5x5 in the entire network. The 9 outputs encode the face, its angle and pose (each output is a 113x153 image).

Our architectural template for the CNN coprocessor consists of an array of convolver primitives connected to external memory. Each convolver primitive can convolve one image with one kernel. The number of convolver primitives, the number of memory ports and the port width are pre-specified for an architecture. The hardware also has no internal storage and requires that all convolver primitives are fed data continuously in a streaming manner. For hidden layers, the values of the hidden units are the intermediate outputs. If the computing architecture has fewer than *n* convolvers per output then accesses to off-chip memory are required to store intermediate data. This is because the computing architecture has no internal storage for intermediate data.

Given this architectural template, there are different ways of organizing the convolver primitives within the memory port (and port width) constraints. We show that the best performance is achieved when the layers use different organizations. We first note that all layers have fine-grained parallelism within a convolution operation. However, a software implementation of a CNN when run on a processor cannot effectively leverage that parallelism due to thread synchronization. Our convolution primitives exploit the parallelism inherent in a convolution using well-known systolic architectures.

Of more interest are the inter-output and intra-output parallelisms. The first layer, which produces 8 outputs from a single input image, exhibits inter-output parallelism, while the second layer exhibits intra-output parallelism. Let us assume the hardware has 8 convolver primitives, 2 memory ports of width 8 pixels (i.e., the ability to transfer 8 pixels per port each cycle). Consider the first layer. It has 1 input image and 8 outputs. If we organize the 8 available convolver primitives so that we have a single broadcast of the input image and the computation of the 8 outputs in parallel, then the computing architecture executes the first layer with maximum performance. However the same arrangement will work poorly for the second layer which has 8 input images and 20 output images. With a single broadcast input and 8 convolvers, we can only process one input at a time, but compute 8 partial outputs in parallel. With no intermediate storage, the 8 partially computed outputs must be written out to off-chip memory resulting in an increase in intermediate data memory traffic. Furthermore, each input image, which participates in producing 20 outputs, will have to be read three times from the off-chip memory in order to generate and write out all the 20 outputs. The execution time is roughly the same as the time it takes to read one of the images 24 times.

On the other hand, assume that the second layer had a different arrangement of the 8 convolver primitives where the 8 convolvers simultaneously process 8 different input images, but the results of the 8 convolutions are combined to produce a single output. With 8 convolvers processing eight different input images,

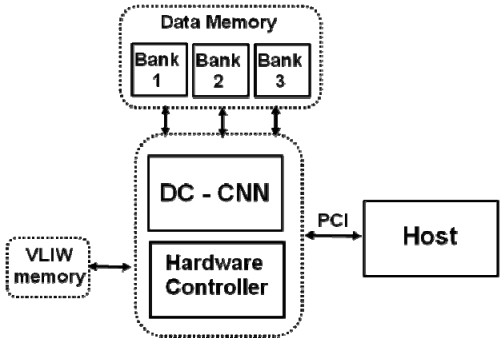

Figure 3: System Overview.

we are computing each of the 20 outputs in a serial fashion. However, each output is being computed in parallel (8 images are being processed in parallel to produce an output). Note that there is no intermediate data to be written off-chip since no partial outputs are computed. This is intra-output parallelism. The execution time for the second layer is roughly the same as the time it takes to read an image, about 20 times. Clearly, the execution time for the second layer is much shorter than the case where the 8 convolvers are processing the same image but producing 8 different (partial) outputs. As shown in the experimental results section, the performance improvement could be as much as 3X for CNN layers in several real-world CNN applications.

5. CNN COPROCESSOR ARCHITECTURE

Our architectural template primarily consists of a processing core and a memory sub-system. The processing core itself is stateless, and we require that input, intermediate and output data are continuously streamed between the processing core and the memory sub-system.

5.1 System Overview

As shown in Figure 3, the processing core DC-CNN communicates with a separate processor ("Host") that executes the main application. The host off-loads the entire CNN computation to the co-processor. In particular, the host transfers the input images and the detailed CNN network structure to the co-processor (number of layers, kernels used in each layer, bias values, sampling kernels, non-linearity etc.). The coprocessor has access to 3 external memory banks (Data Memory) to store input images, kernels and intermediate data.

We have developed a CNN compiler (it executes on the host platform) that automatically translates high abstraction network specification written by domain experts into a parallel microprogram (a sequence of low-level VLIW instructions) that is mapped, scheduled and executed by the coprocessor (hardware controller in Figure 3). Instructions to facilitate dynamic configurability, complex control and data flows, as well as on-the-fly packing of intermediate data to minimize off-chip memory transfers, are also natively supported by the coprocessor.

The co-processor performs forward propagation across all the network layers and returns the output of the last layer back to the host. We chose a high-level of abstraction for the host interface since implementing only the convolution task on the co-processor will require the host to co-ordinate complex control and data flows, and this will severely degrade the performance. Also, significant amounts of intermediate data are generated within and

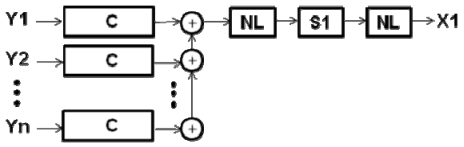

Figure 4: Basic computational element: hardware pipeline of convolvers (C), non-linearity (NL) and sub-sampling (S1).

across layers of the CNN. Moving the intermediate data across the slow host-coprocessor interface will negate any advantage we get from performing fast convolutions on the co-processor.

5.2 The Processing Core

Each layer of the CNN reads one or more images as input, and computes one or more images as output. Our basic computational element, shown in Figure 4, is designed to compute one output image at a time. This computational element is architected to take advantage of intra-output parallelism. The basic computation primitive is the 2D convolver that can store a kernel internally. We use a bank of 2D convolvers whose outputs are aggregated. The computation element also has a specialized hardware pipeline to compute non-linearity (NL) and sub-sampling (S1). If more than ***n*** images have to be combined to realize an output, then the aggregated output from the convolver bank may only be a partial output (i.e. intermediate data) that must be stored in off-chip memory. Partial outputs are not subjected to non-linearity or sub-sampling.

If the coprocessor reads one pixel every cycle, then the time taken for reading the input is the same as the total number of pixels in the input images. If the memory architecture is designed so that the co-processor can input multiple pixels in one cycle, then the time taken for reading the input information will reduce proportionally. Either way, every pixel in every input image has to be examined at least once to compute an output image. Fetching data from off-chip memory consumes significantly more power than typical operations performed within the processor. Therefore, minimizing memory transfers has a significant effect on power consumption. One of our key design goals is to read each input pixel exactly once, and compute the output image within a few (fixed number of) cycles after the last input pixel is read. By computing the output image in approximately the same time it takes to read input data, the basic computational element is optimal with respect to a given off-chip memory bandwidth, power consumption, and throughput.

If we use multiple computational elements, each input image can be simultaneously used to compute more than one output image. Since the same image is used to compute different output images, we can fetch the input image only once. Figure 5 shows

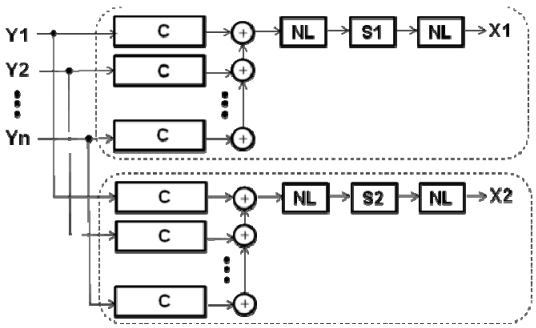

Figure 5: Processing core with two computational elements.

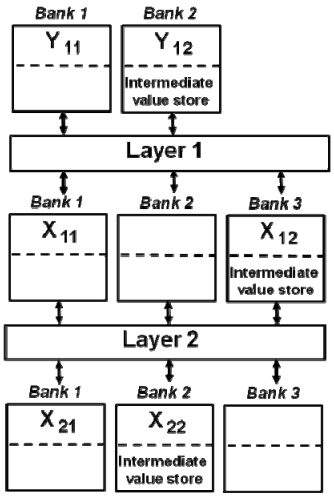

Figure 6: Use of three memory banks.

an array of two computational elements (X_1 and X_2) where the inputs of the first computational element are also broadcast to the second element. With such a configuration, we can leverage inter-output parallelism, since we can now compute two outputs simultaneously.

By varying the number of convolvers in a computational element (***Y***), as well as the total number of computational elements (***X***), we can control the extent to which the two different parallelisms are parlayed to match the exact computational workload of any layer of a CNN. Note that by replacing the convolver primitive with a different functional unit, we can easily implement different feed-forward neural networks and classifiers such as HMAX [17][27], Deep Belief Networks [29][30] and HoG methods [28].

5.3 The Memory Sub-system

Design of the memory subsystem has a big impact on the performance of CNNs as well as other artificial neural networks. Since our processing core is stateless, multiple banks that can be simultaneously read from and written to are necessary. A banked memory subsystem is indispensible for CNNs rather than a single memory with the same or higher aggregate bandwidth and storage. For the discussion in this section, we assume that the memory subsystem consists of one or more banks, and each bank has exactly one single-ported memory. However, basic ideas described in this section can be easily adapted to multi-port memories. If ***Y***, the number of convolvers in a computational element, is less than I_n, then we need multiple passes to compute each output, thereby resulting in intermediate data values. We assume that each memory bank has the necessary port width to either read ***Y*** input pixels (and if necessary, ***X*** intermediate data values for the ***X*** output images being computed simultaneously) and write ***X*** output pixels (or ***X*** intermediate data values), every cycle.

In this section, we show that three independent memory banks suffice to ensure uninterrupted data flow for the stateless CNN architecture. Consider Figure 6. Layer 1 of a CNN produces 2 outputs (X_{11} and X_{12}) from 2 input images (Y_{11} and Y_{12}). Also, assume that we choose a co-processor architecture that has ***Y*** =1 and ***X*** = 1 (i.e. one 2D convolver, and one NL-SS-NL pipeline). Since ***Y*** = 1, we process one image at a time. Every cycle, we read a pixel from an input image. We either generate an intermediate value or we compute a pixel of one of the output images, but not both.

To compute the output image X11, we first read pixels of image Y_{11} that is stored in Bank 1. Processing of every pixel in Y_{11} generates one intermediate value. Since we are reading a pixel from Bank 1 in every cycle, we cannot also write the intermediate value to Bank 1 in the same cycle. Furthermore, while reading pixels of image Y_{12}, we must also read the intermediate values. Therefore, intermediate values and pixels of Y_{12} are laid out in the same memory bank (Bank 2). Convolution of image Y_{12} will not result in intermediate values since this is the last image. Since X_{11} cannot be written to Bank 2, we store X_{11} in Bank 1. After we compute X_{12}, where do we store X_{12}? Like Layer 1 where the two input images Y_{11} and Y_{12} were in different banks, Layer 2 processing dictates that X_{11} and X_{12} be stored in different memory banks. Therefore, we store the output X_{12} in a new bank (Bank 3). Clearly, three memory banks are necessary and sufficient to support stateless processing. Output images of Layer 1 are the input images for Layer 2, and we can argue similarly to determine the location of Layer 2's output images (X_{21} and X_{22}).

6. DYNAMIC CONFIGURATION

In order to make the core adapt to the different types of parallelism across CNN layers, we introduce an input switch as shown in Figure 7. The switch allows the convolvers to be grouped in different ways by varying ***Y*** and ***X.*** The maximum values of ***Y*** and ***X*** are determined by the memory bandwidth and the area budget of the chip. Assuming a 3-bank architecture, if the memory bandwidth per bank is ***P*** bits per cycle, the input pixel width is I_b bits, intermediate data width T_b and output pixel width O_b, then the maximum value of ***Y***, given by Y_{max} is $\left\lfloor \min\left(\frac{P}{Ib}, \frac{P}{Tb}\right)\right\rfloor$. Similarly, the maximum value of ***X*** (X_{max}) is $\left\lfloor \min\left(\frac{P}{Ob}, \frac{P}{Tb}\right)\right\rfloor$.

Note that every computational element has to sum the ***Y*** convolutions prior to non-linearity and sub-sampling. Since ***Y*** can change in the configurable architecture, this summation is achieved by using a sea of adders each statically designed to add Y_{max} values. Specifically, we need X_{max} adders, each with Y_{max} inputs. The output switch appropriately routes the ***Y*** * ***X*** convolver outputs to appropriate inputs of these adders. The routing is done in a block-wise fashion, i.e., groups of ***Y*** outputs from the convolvers are routed to a single adder with Y_{max}-inputs.

6.1 Input Switch and Output Switch

Figure 8 shows the details of the input and output switch for a design where Y_{max} = 6 ***and*** X_{max} = 2. The input switch is constructed from a simple selector (multiplexor). The selector has three inputs (one select signal and two input values, one of which is 0). Depending on the value of the 'Select' signal, the selector either produces a 0 value as output or it forwards the input value. Each selector is connected to exactly one convolver. We achieve dynamic configurability as follows. For a CNN layer with 3 input images (Y_1, Y_2 and Y_3), and one output image X_1, we generate select signals so that the first three selectors of computational element X_1 (selectors M_1, M_2 and M_3) are enabled to forward their inputs (images Y_1, Y_2 and Y_3) to their convolvers. Selectors M_4, M_5 and M_6 are configured to forward a value of 0. So, these selectors are effectively *disabled*. Please note that the 2D convolvers are also designed to perform no computations if their corresponding selector is disabled (this is not required for correctness of the result, but it saves the power that would otherwise be consumed by the convolver). Similarly, the first three selectors of

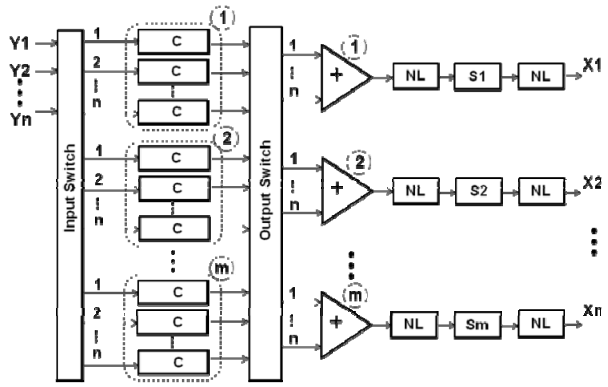

Figure 7: Dynamically configurable CNN architecture.

computational element X_2 (M_{19}, M_{20} and M_{21}) are configured to forward images Y_1, Y_2 and Y_3 to their convolvers. Selectors M_{22}, M_{23} and M_{24} are disabled.

In a simple implementation of the output switch, we chain the convolvers in a computational element to realize the pipelined summation of the convolver outputs. For example, selectors M_1, M_2 and M_3 are set to forward input images Y_1, Y_2 and Y_3 to their convolvers. Outputs of the three convolvers are aggregated over time as shown in Figure 8. The aggregated value is then processed by the non-linearity and sub-sampling units.

6.2 Exploring Architectural Configurations

Scaling the number of processing elements in a chip is easier than scaling off-chip memory bandwidth. Therefore, in practice, finding the optimal values of ***Y*** and ***X*** for a CNN layer boils down to finding a judicious mix of intra-output and inter-output parallelism that saturates the available off-chip bandwidth.

We first examine constraints on ***Y*** and ***X,*** assuming a 3-bank memory subsystem where each bank transfers **P** bits per cycle. Again, assume that the input pixel width is $\mathbf{I_b}$, the intermediate data width is $\mathbf{T_b}$, the output pixel width is $\mathbf{O_b}$ and that there are **C** convolvers in the hardware. The product of ***Y*** and ***X*** *must* be less than or equal to the total hardware available which means $\boldsymbol{X*Y \leq C}$. Several constraints are due to the memory sub-system. Specifically, it is straightforward to see that:

$$\boldsymbol{X * I_B \leq P} \text{ (if X is used to write intermediate data)}$$

$$\boldsymbol{X * O_B \leq P} \text{ (if X is used to write final outputs)}$$

$$\boldsymbol{Y * T_B \leq P} \text{ (if Y is used to read intermediate data)}$$

$$\boldsymbol{Y * I_B \leq P} \text{ (if Y is used to read first layer inputs)}$$

When computing a CNN layer, if we only read primary inputs (and no temporary intermediate result), the number of memory bits read per cycle is $\boldsymbol{Y * I_B}$. If there are temporary results, there are $\boldsymbol{(Y * I_B) + (X * T_B)}$ bits read per cycle. Similarly, the number of memory bits written per cycle is $\boldsymbol{X * T_B}$ or $\boldsymbol{X * O_B}$ depending on whether we store intermediate or final outputs. Read requests per cycle or write requests per cycle cannot exceed ***P*** bits per cycle.

For a CNN layer with *n* inputs and *m* outputs, the number of passes required for completion is $\boldsymbol{(n/Y)*(m/X)}$. The time for completing one pass is simply the time required to fetch the image from memory one pixel at a time (since we have a stateless streaming architecture): $I_r * I_c$ where I_r is the number of image rows, and I_c the number of image columns. Therefore, the execution time for completing the entire layer is $\boldsymbol{(I_r * I_c) * n /Y *}$ $\boldsymbol{m /X}$. This is the cost function that we want to minimize subject to the memory sub-system constraints, and total hardware available (number of convolvers). From a system point of view, a run-time software component analyzes the CNN workload, and determines the best ***(Y,X)*** for each layer. The CNN architecture is then dynamically configured for each layer by using a special instruction in the instruction set of the coprocessor.

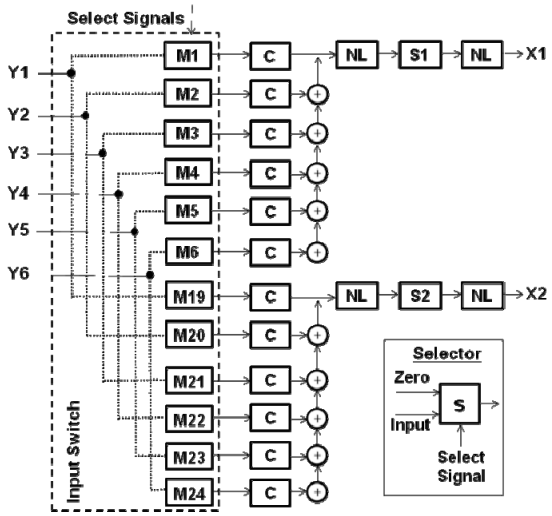

Figure 8: Input and output switches.

The algorithm for choosing the best ***(Y, X)*** for each layer uses integer factorization techniques to identify different candidate integer values of ***Y*** and ***X*** so that $\boldsymbol{X*Y \leq C}$. For each value of ***Y*** and ***X,*** we evaluate the memory sub-system constraints to determine if the specific combination of ***Y*** and ***X*** is a feasible solution. We compute the cost function for every feasible solution, and select the feasible solution that minimizes the cost function (execution time for processing the entire layer). Although no efficient integer factorization algorithm is known for factoring very large numbers (with 100 to 200 digits), there are several reasons why this approach works well in practice. First, the largest number of hardware convolvers (***C***) we considered is 40. Even if ***C*** increases by two or three orders of magnitude (very unlikely due to power and hardware constraints), fortunately, integer factorization for small numbers is fast [10]. Second, by using dynamic programming, we are able to quickly prune many combinations of ***Y*** and ***X.*** For example, consider the case of C = 40 convolvers, and a memory port-width of 128 bits of data per cycle. To determine the best configuration for Layer 3 of the Video Surveillance workload, only a small number (14) of feasible combinations, shown in Figure 9, had to be considered.

7. EXPERIMENTAL RESULTS

In this section, we first compare the performance of a fixed coprocessor and a dynamically reconfigurable coprocessor using several real CNN workloads. We then examine the scalability of the new architecture with respect to scaling (increase) in number of processing elements, as well as memory bandwidth, and show that dynamic configurability has a consistent, first-order effect on performance. Finally, we compare the performance of our dynamically configurable CNN coprocessor with leading CNN workload implementations reported recently on several other computing platforms.

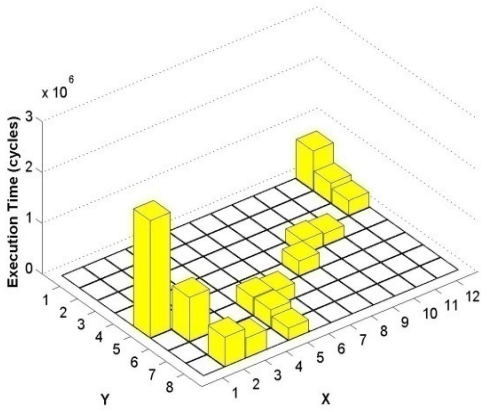

Figure 9: Feasible architectural configurations for Layer 3 of the Video Surveillance workload.

We estimated performance and scalability using a simulator that provides cycle-accurate execution time estimates of the architectural components. In the simulator, ***Y*** and ***X*** could be parameterized per layer or they can be fixed for the entire CNN workload. Our architectural simulator uses an emulation of a 20-convolver, 3-memory bank (128 bits of data per cycle per port per bank), dynamically configurable CNN architecture (including the hardware learning engine controller) on a Virtex 5 SX240T FPGA platform (1024 multiply-accumulate units) to more accurately estimate execution cycles. The RTL (Verilog) for the 20-convolver design was mapped to the FPGA by using design tools from Synplicity. In all CNN workloads, an input or output value is represented using 16 bits, and an intermediate data value is represented using 48 bits. All CNN workloads are specified by using a simple, high abstraction level software programming API. A CNN compiler (running on the host processor) automatically translates the entire high abstraction network specification into a parallel micro-program (a sequence of low-level VLIW instructions) that is mapped, scheduled and executed on the coprocessor FPGA implementation. Instructions to facilitate complex control and data flows, as well as on-the-fly packing of intermediate data to minimize off-chip memory transfers, are also natively supported by the coprocessor.

7.1 Workloads

Table 1 shows the five different CNN workloads we considered. In the "Video Surveillance" workload that was developed internally, the CNN is used to identify human activity within a video frame and to recognize specific physical characteristics such as age, gender and race. The "Face Recognition" workload is obtained from [26]. The "Automotive Safety" workload is a vision, range and motion sensing network, used in automotive safety applications, to process video-feeds from vehicle-mounted cameras. This application was developed internally for a major Japanese automobile manufacturer. The "Mobile Robot Vision" workload is obtained from [9]. Here, a CNN is used for learning long-range vision (on-vehicle Robot) for autonomous, off-road driving. The "Face Detection" workload is obtained from [25]. As shown in Table 1, the CNNs used in these applications have a wide range of inputs and outputs across different layers. All examples use VGA frames (640 x 480 pixels per frame).

7.2 Advantage of Dynamic Configurability

Table 2 compares the dynamically configurable architecture with the best fixed-architecture, in terms of execution cycles. The fixed architecture has the same **(*Y, X*)** for all three layers, while the dynamically configurable architecture adapts to each layer of the CNN. For all workloads, the hardware constraint is 20 convolvers, and the memory port-width is 128 bits of data per cycle. The column "Best Fixed Architecture" reports the values of ***Y*** and ***X*** that were chosen, and the total execution cycles for each of the workloads. Note that the ***Y*** and ***X*** values are fixed for all three layers of a CNN. The column "Dynamically configurable" reports (a) the values of ***Y*** and ***X*** that were chosen for each layer of a CNN workload, and (b) the total execution cycles for all three layers of a CNN workload. The last column reports the speedup of the dynamically configurable version as compared with the fixed architecture case. Our results show that a dynamically configurable architecture consistently outperforms a fixed architecture. Depending on the CNN workload, speedup factors range from 1.5x to 2.3x.

To understand the scalability of the new architecture with respect to scaling (increase) in number of processing elements, as well as increase in memory bandwidth, we considered four CNN coprocessor architectures with 10, 20, 30 and 40 convolvers. In addition, we considered three different memory port widths (64 bits per cycle, 128 bits per cycle and 256 bits per cycle) for each of the four CNN architectures. For all architectures, we used 3 independent memory ports. Figure 10, Figure 11 and Figure 12 show the speedup obtained for three different CNN workloads, as a function of the number of convolvers and memory port width (due to space constraints, we are not able to include data for all experiments). For example, consider the case of the "Automotive Safety" workload. We designed the best possible fixed architecture, assuming a hardware constraint of 10 convolvers, and a memory port width of 64 bits of data per cycle. In order to obtain the best fixed architecture, we examine all possible valid combinations of ***Y*** and ***X***, and select the best one for a fair comparison. Then, we designed a dynamically configurable architecture under the same hardware and memory constraints. From the figures, we observe that the dynamically configurable version is about 2.2x faster than the best fixed-architecture version (for 10-convolvers, and 64-bit memory port width). Similar

Table 1 : CNN workload characteristics.

	Video Surveillance		**Face Recognition**		**Automotive Safety**		**Mobile Robot Vision**		**Face detection**	
	Inputs	Outputs	Inputs	Outputs	Inputs	Outputs	Inputs	Output	Inputs	Outputs
Layer 1	5	6	1	20	1	8	1	6	1	6
Layer 2	6	16	20	25	8	20	6	16	6	16
Layer 3	16	120	25	40	8	20	16	80	16	100

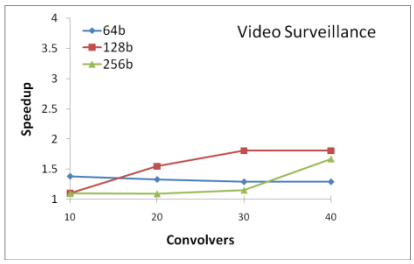

Figure 11: Speedup for Video Surveillance.

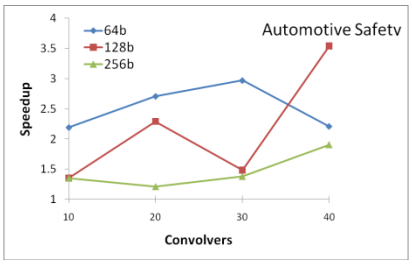

Figure 10: Speedup for Automotive Safety.

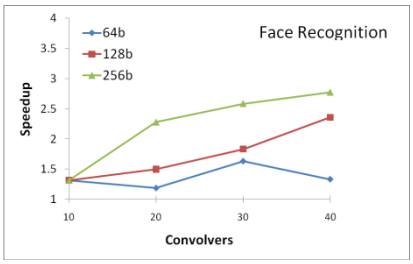

Figure 12: Speedup for Face Recognition.

experiments were conducted to obtain data for all the four architectures and three different memory port widths per architecture.

We note three interesting trends from the data. First, irrespective of the number of convolvers or the memory port width, the dynamically configurable version is consistently faster (speedups range from 1.2x to 3.4x) than the corresponding, best fixed-architecture version. Second, increasing the number of convolvers (for a given memory port width) usually results in higher speedups. However, in some cases the best fixed architecture also happens to be a reasonably good match for the given workload, resulting in an occasional low speedup factor for the dynamically configurable version. For example, consider the 128-bit memory port width case for the "Automotive Safety" workload. With 30 convolvers, the dynamic version is only 1.5x faster than the best fixed architecture (for a 20-convolver design, the dynamic version is 2.3x faster than the best fixed architecture for the same workload). Third, increasing the memory port width (for a given number of convolvers) may or may not result in higher speedups. It does improve both designs (fixed or dynamically configurable) but the amount of improvement can be very different. For some workloads like the "Face Recognition" workload, with large amount of intermediate data across the three layers, the advantage of dynamic configurability is more pronounced at wider memory port widths. We see the exact opposite effect for the "Automotive Safety" workload, where higher speedups are usually achieved for smaller memory port widths. The absolute performance of fixed or dynamically configurable architecture always improves with higher memory port widths, and so does the power consumption. In all cases, the utilization factor of the hardware convolvers or the available memory bandwidth is close to 100%.

7.3 Comparison with Other Implementations

We also compared the performance of our dynamically configurable CNN coprocessor (20 convolvers, 128-bit memory port width) with leading CNN workload implementations reported recently on several platforms: (a) 128-core, 1.35GHz NVIDIA's GPU with 1.5GB RAM, and a fast PCI Express connection to the x86 host [26] (b) an internal software implementation on an dual-socket, quad-core, 2.33 GHz Intel Xeon (Intel Multicore with eight 2.33 GHz cores), and (c) a 200 MHz, fixed architecture CNN co-processor called CNP that was implemented on a Virtex 4 FPGA part from Xilinx [9].

Results for the Intel Xeon (8-processor) multicore (column "Multicore" in Table 3) were obtained by using an internal software implementation of CNN on the multicore. This implementation uses the latest BLAS library that has been specially optimized for dense linear algebra operations. The performance of the multicore implementation is quite competitive with the GPU implementations. GPU results reported in [26] used an older, 600 MHz NVIDIA GPU. They report a processing time of 210 ms per frame for the "Face Detection" workload. Their frame has less pixels than our VGA frame. On a faster GPU (1.35GHz), we achieve a processing time of 105 ms per VGA frame. If we factor in the difference in clock speeds, the two GPU implementations are comparable. We use our GPU implementation to generate results for all the other workloads.

Performance of software CNN implementations on embedded processors is poor. For example, processing times per frame on the Intel Atom processor were 1.67 s (Automotive Safety), 2.05s (Video Surveillance), 1.95s (Face Recognition), 1.23 s (Mobile Robot Vision), and 0.97s (Face Detection). We see a clear 10X pullback from software implementations on the Xeon.

Results for the 200 MHz CNP (column "CNP" in Table 3) were taken from [9]. They report a processing time of 100 ms per frame for the "Mobile Robot Vision" workload. Since they do not use a separate host processor, there is no need to transfer the video feed from the host. Also, we do not have access to their hardware, and we are unable to report results for the remaining workloads on their platform.

Results for the proposed dynamically configurable CNN coprocessor (column "DC-CNN" in Table 3) were obtained using

Table 2 : Comparison of fixed and dynamically configurable, 20-convolver, 128-bit memory port width.

CNN workload	Best Fixed Architecture		Dynamically configurable		Speedup
	Y,X	Performance Cycles	***Y,X***	Performance Cycles	
Automotive Safety	***7,2***	3,340,800	L 1: ***1,8*** L 2: ***8,2*** L 3: ***10,2***	1,518,545	2.2x
Video Surveillance	***7,2***	4,972,000	L 1: ***5,4*** L 2: ***4,5*** L 3: ***10,2***	3,225,600	1.5x
Face Recognition	***8,2***	7,603,200	L 1: ***1,8*** L 2: ***8,2*** L 3: ***10,2***	5,068,800	1.5x
Mobile Robot Vision	***10,2***	3,072,000	L 1: ***1,6*** L 2: ***10,2*** L 3: ***10,2***	2,457,600	1.3x
Face Detection	***8,2***	3,456,000	L 1: ***1,6*** L 2: ***8,2*** L 3: ***10,2***	2,841,600	1.2x

Table 3: Comparison with other CNN implementations.

CNN (640 x 480 pixels input image)	**Multicore** (Xeon @ 2.33 Ghz, 8 Cores, 16 GB) BLAS	**GPU** (C870 @ 1.35 Ghz, 1.5 GB RAM) PCIe	**CNP** (FPGA @200 MHz)	**DC-CNN** @ 120 Mhz 20 conv., 128-bit port width, PCI		**Speedup** of DC-CNN		
				Compute time	Transfer time	Over 2.3 GHz, 8-core	Over 1.35 GHz, 128-core GPU	Over CNP
Automotive Safety	110 ms	85 ms	-	13 ms	11 ms	8.5x	6.5x	-
Video Surveillance	212 ms	163 ms	-	27 ms	34 ms	7.8x	6.0x	-
Face Recognition	217 ms	167 ms	-	42 ms	11 ms	5.2x	4.0x	-
Mobile Robot Vision	147 ms	114 ms	100 ms	21 ms	11 ms	7.0x	5.4x	4.8x
Face Detection	136 ms	105 ms	-	24 ms	11 ms	5.7x	4.4x	-

a 120 MHz FPGA implementation of our dynamically configurable co-processor, with 20 convolvers and 128-bit memory port width. We report separately the time taken to transfer the images (from the x86 host to the coprocessor) and output the results (from coprocessor to the host). In our implementation, transfer of images or output results overlaps with the CNN computation. Therefore, the total processing time for any CNN workload is the maximum of the execution time of the coprocessor and the data transfer time. For example, consider the case of the "Face Recognition" workload. It takes 11 ms to transfer the input image data to the co-processor, and it takes 42 ms to perform the processing for all three Layers in the CNN workload. However, processing of a Layer begins as soon as portions of images are available on the coprocessor.

Similarly, some of the output results computed by the coprocessor are transferred to the host while other outputs are still being computed by the co-processor. Therefore, due to the overlap, the total time required to process the frame was only 42 ms. Column "Speedup" in Table 3 shows the performance advantage of the dynamically configurable architecture over the multicore, GPU and CNP. We observe that the dynamically configurable architecture is 4x to 8x faster. More importantly, the additional speedup of the dynamically configurable architecture now enables an important tipping point. *By easily processing 25 to 30 frames per second, the proposed dynamically configurable co-processor is the first CNN architecture to enable real-time video stream processing on a wide range of object detection and recognition tasks.* Due to clock frequencies in the GHz range, the GPU and the Intel Multicore implementations consume more than 150 Watts. The 200 MHz Virtex4 implementation consumes 15 Watts [9]. In comparison, our 120MHz dynamically configurable co-processor prototype on a Virtex 5 FPGA consumes less than 14 watts of power (for all components on the FPGA board, including memory banks).

8. CONCLUSIONS

We presented a dynamically reconfigurable architecture for feed-forward neural networks used in recognition, analysis and reasoning applications. The design of the architecture is driven by two key observations. The first is based on the fact that CNNs exhibit "inter-output" and "intra-output" parallelism. Inter-output parallelism is where different outputs can be computed independently, and in parallel. Intra-output parallelism exploits parallelism within a single output computation. We showed that different CNN workloads exhibit a widely varying mix of these two types of parallelism within a single network. That is, different layers of a network must be parallelized in different ways. We therefore argued that the architecture itself must adapt to the way a particular layer of a network needs to be parallelized. This adaptive architecture is achieved by allocating an array of convolver primitives and dynamically configuring their organization at run-time to achieve optimal performance. The second observation is based on the fact that CNNs have large intermediate data which cannot be stored on-chip. Therefore, we designed a streaming architecture with multiple memory ports where input data, intermediate data and output data are continuously flowing between the processor and off-chip memory. We showed that a dynamically configurable architecture can provide speedups ranging from 1.2x to 3.5x over a similar fixed custom architecture with the best possible fixed configuration of the convolver primitives.

REFERENCES

[1] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, (pp. 1-46).

[2] Collobert, R.; Weston, J., "A unified architecture for natural language processing: deep neural networks with multitask learning," *Proceedings of the 25th International Conference on Machine Learning* (*ICML 2008)*, vol. 307, pp.160-167, Jul 2008.

[3] Benkrid, K.; Belkacemi, S., "Design and implementation of a 2D convolution core for video applications on FPGAs," *Digital and Computational Video, 2002. DCV 2002. Proceedings. Third International Workshop on*, pp. 85-92, 14-15 Nov. 2002.

[4] Cardells-Tormo, F.; Molinet, P.-L., "Area-efficient 2-D shift-variant convolvers for FPGA-based digital image processing," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol.53, no.2, pp. 105-109, Feb. 2006.

[5] Hui Zhang; Mingxin Xia; Guangshu Hu, "A Multiwindow Partial Buffering Scheme for FPGA-Based 2-D Convolvers," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol.54, no.2, pp.200-204, Feb. 2007.

[6] Savich, A.W.; Moussa, M.; Areibi, S., "The Impact of Arithmetic Representation on Implementing MLP-BP on

FPGAs: A Study," *Neural Networks, IEEE Transactions on*, vol.18, no.1, pp.240-252, Jan. 2007.

[7] Gironés, R. G.; Palero, R. C.; Boluda, J. C.; Cortés, A. S., "FPGA Implementation of a Pipelined On-Line Backpropagation," *J. VLSI Signal Process. Syst.,* vol. 40, no. 2, pp.189-213., Jun 2005.

[8] Catanzaro, B.; Sundaram, N.; Keutzer, K., "Fast Support Vector Training and Classification on Graphics Processors," *Machine Learning, 25th International Conference on, (ICML 2008)*, Jul. 2008.

[9] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based Processor for Convolutional Networks", in *Proc. International Conference on Field Programmable Logic and Applications (FPL'09)*, IEEE, Prague, 2009.

[10] Dixon, J. D. (1981). Asymptotically fast factorization of integers. *Math. Comput. , 36*, 255-260.

[11] Hadsell, R. e. (2009). Learning long-range vision for Autonomous off-road Driving. *Journal of Field Robotics , 26* (2), 120-144.

[12] Haykin, S. (2008). *Neural networks and learning machines.* Prentice Hall.

[13] Korekado, K., Morie, T., Nomura, O., Nakano, T., Matsugu, M., & Iwata, A. (2005). An Image Filtering Processor for Face/Object Recognition using Merged Analog-digital architecture. *Symposium on VLSI Circuits*, (pp. 220-223).

[14] Lisboa, P., Ifeachor, E., & Szczepaniak, P. (2009). *Artificial neural networks in Biomedicine.* Springer

[15] McNelis, P. D. (2005). *Neural Networks in Finance: Gaining Predictive Edge in the Market.* Academic Press.

[16] Mirowski, P. e. (2008). Comparing SVM and Convolutional networks for Epileptic Seizure Prediction from Intracranial EEG. *Proceedings of Machine Learning and Signal Processing*, (pp. 244-249).

[17] Mutch, J., & Lowe, D. (2006). Multiclass object recognition with sparse, localized features. *International Conference on Computer Vision and Pattern Recognition*, (pp. 11-18).

[18] Nakajima, M., & al., e. (2006). A 40GOPS 250mw massively parallel processor based on matrix architecture. *International Solid-state Circuits Conference*, (pp. 410-411).

[19] Nichols, K., Moussa, M., & Areibi, S. (2002). Feasibility of floating-point arithmetic in FPGA based artificial neural networks. *Proceedings of the 15th International Conference on Computer Applications in Industry and Engineering.* San Diego, California

[20] Nomura, O., & Morie, T. (2007). Projection-Field-Type VLSI Convolutional Neural Networks Using Merged/Mixed Analog-Digital approach. *International Conference on Neural Information Processing* (pp. 1081-1090). Springer-Verlag.

[21] Omondi, A., & Rajapakse, J. (2006). *FPGA Implementations of Neural Networks.* Springer.

[22] Prasad, B., & Prasanna, S. (2008). *Speech, Audio, Image and Biomedical Signal Processing using Neural Networks.* Springer.

[23] Sermanet, P. e. (2009). Multi-range architecture for collision-free off-road Robot Navigation. *Journal of Field Robotics , 26* (1), 58-87.

[24] Wolf, D. F., Romero, R. A., & Marques, E. (2001). Using embedded processors in hardware models of artificial neural networks. *Proceedings of SBAI - Simposio Brasileiro de Automao Inteligente*, (pp. 78-83).

[25] Steve Lawrence, C. Lee Giles, Ah Chung Tsoi, Andrew D. Back, Face Recognition: A Convolutional Neural Network Approach. IEEE Transactions on Neural Networks 1997.

[26] Nasse, F., et al , "Face Detection using GPU-based Convolutional Neural Network", CAIP 2009, LNCS pp 83-90, Springer Verlag

[27] Serre, T. et al "Object recognition with features inspired by the visual cortex", Proceedings of Computer Vision and Pattern Recognition 2006.

[28] Dalal, N. et al, "Histograms of oriented gradients for human detection", Proceedings of Computer Vision and Pattern Recognition, 2005

[29] Raina, R. et al, "Large-scale Deep Unsupervised Learning using Graphics Procesors", Proceedings of International Conference on Machine Learning, 2009 (pp. 873-880).

[30] Lee, H. et al, "Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations", Proceedings of International Conference on Machine Learning, 2009 (pp. 873-880).