An Efficient Implementation of Deep Convolutional Neural Networks on a Mobile Coprocessor

Jonghoon Jin*, Vinayak Gokhale*, Aysegul Dundar†, Bharadwaj Krishnamurthy*,
Berin Martini† and Eugenio Culurciello*†
*Electrical and Computer Engineering, Purdue University
†Weldon School of Biomedical Engineering, Purdue University

***Abstract*—In this paper we present a hardware accelerated real-time implementation of deep convolutional neural networks (DCNNs). DCNNs are becoming popular because of advances in the processing capabilities of general purpose processors. However, DCNNs produce hundreds of intermediate results whose constant memory accesses result in inefficient use of general purpose processor hardware. By using an efficient routing strategy, we are able to maximize utilization of available hardware resources but also obtain high performance in real world applications. Our system, consisting of an ARM Cortex-A9 processor and a coprocessor, is capable of a peak performance of 40 G-ops/s while consuming less than 4 W of power. The entire platform is in a small form factor which, combined with its high performance at low power consumption makes it feasible to use this hardware in applications like micro-UAVs, surveillance systems and autonomous robots.**

I. Introduction

Artificial vision systems find applications in autonomous robots, security systems, micro-UAVs and more recently, mobile phones and automobiles. These applications require algorithms that can recognize objects with a high degree of accuracy while being able to execute in real-time. Many recent algorithms have shown promise for use in visual understanding. For example, SIFT and SURF feature extractors [1], [2], hierarchical models of the visual cortex (HMAX) [3] and deep convolutional neural networks (DCNNs) [4]–[6] can perform robust feature extractions for recognition, detection and scene understanding.

Recently DCNNs, which consist of hundreds of convolution planes across multiple layers, have shown great promise in visual understanding [4] as well as artificial intelligence [7]. However, DCNNs are computationally very expensive and require high performance computers or graphics processing units (GPUs) to run in real time. For mobile, low-power platforms like smartphones, computations are usually sent to off-site servers systems for processing. This requires constant and reliable connectivity with the off-site server which is not a guarantee. This constraint limits the use of DCNNs in mobile environments where very less lag is allowed between the input and the result, such as when driving automobiles. Mobile processors have becoming increasingly powerful in the last decade. However, their real time performance is application dependent and can be much lower than their peak performance. This is mainly the result of underutilization of hardware resources due cache misses, frequent, unpredictable branches and a large amount of memory accesses. For these reasons, it is difficult to process DCNNs in real-time on current mobile, low-power platforms.

In this work, we present an efficent implementation for accelerating DCNNs on a mobile platform in a pipelined manner. Our system is implemented on the Xilinx Zynq-7000 All Programmable SoC which combines two ARM Cortex-A9 cores with an Artix-7 FPGA. Our hardware implementation allows for maximum utilization of hardware resources which results in real time performance that is very close to peak performance.

This paper presents as follows: Section II explains preliminaries. Section III outlines properties of deep convolutional neural networks. Section IV describes implementation details and strengths of our design, and section V presents experimental results and discussions of our hardware.

II. Related Work

Hardware accelerated vision systems implemented on FPGAs have been described in [8]–[10]. All three works require a host computer with a PCI connection in order to function. These cannot easily be embedded inside of small and lightweight mobile robots. Furthermore, these designs require large off-chip memories of their own as the communication bottleneck over PCI when using the host's memory would decrease performance.

Similar streaming architectures [11], [12] have been designed to meet the computational demands of DCNNs, and [12], [13] demonstrates the application of one such system. Their design on fully programmable logic benefits from flexibility and parallelism while overcoming memory bottlenecks by using the on-board DDR3 memory. However, they suffer from slow host-coprocessor data transfer. The vertical off-board connection used in [12] limits its performance when it comes to real world applications. A tile structure of processing elements in [12], [13] facilitates flexible routing. However, not all connections are essential and this results in unnecessary utilization of FPGA resources. Furthermore, such dense routing is unfeasible in SoCs where chip area is an important factor.

Compared to these platforms, the system described in this work uses the Xilinx Zynq SoC which is a shared memory architecture where the same DDR3 memory can be accessed by both the software and the FPGA. This platform is well suited for frequent memory accesses and requires no other system in order to function. Furthermore, the novelty of this work is the router network that is specifically targeted to process DCNNs. The entire platform measures $16.3cm \times 13.1cm \times 2.6cm$ and consumes 3.9 W making it a small but powerful system that can be easily installed in a mobile platform.

III. Deep Convolutional Neural Networks

Deep convolutional neural networks are a class of models that form a powerful tool to help solve visual classification problems [4]–[6]. DCNNs consist of multiple layers of convolutions, each comprising between tens and hundreds of filters. Each convolution layer is interspersed by one sub-sampling and non-linearity operator. In the first layer, convolutions are used to extract low-level features, such as edges and textures. Deeper layers aim at combining the features extracted by the previous layers to achieve a higher level of abstraction and detect more complex features. After each convolution layer, DCNNs use a spatial pooling operator to provide the network with scale invariance. Spatial pooling also results in subsampling which reduces the number of computations required in latter layers. Finally, a non-linearity operation serves as an activation function that helps the classifier to solve non-linear problems. These operators and their implementations in hardware are defined in section IV.

IV. Hardware Implementation

Our hardware comprises a memory router and the operators needed for processing DCNNs as shown in figure 1. The operators are bundled together into a unit called a collection. Each collection also includes a collection router. These two routers are explained in detail in the subsections below.

A. Memory Router

The memory router interfaces with three AXI DMA engines and acts as a gateway to the collections. All incoming and outgoing streams pass through this router. It can be configured to route incoming data streams to one or more collections. The memory router can sustain data transfer speeds of 2.2 GB/s. Routing paths are configured via the configuration bus which is indicated in dark blue in figure 1. This bus is directly controlled by the ARM through a custom device driver. A software compiler allocates hardware resources to the data streams. The configuration order for an entire frame is generated at compile-time. However, due to the limited availability of resources, configuration of the hardware is done at run-time.

B. Collection Network

An efficient design of memory routing is important for large-scale processing systems since such are often limited by memory bandwidth [14]. The novelty of this design is the network of collections and their routing strategies. The collection is the basic processing element in our design that can be run in parallel. The collection network is comprised of collections and data paths around them. The router's design allows for maximum utilization of each collection resulting in a peak performance of 20 G-ops/sec per collection.

Each collection contains a group of operator blocks that performs arithmetic operations. This structure is suitable for DCNNs because these networks typically follow a sequence of processing steps. Thus, these operators need to be spatially grouped together in order to avoid unnecessary routing delays of data streams. This design produces an output of a 1-to-1 convolution plane by a single pass through a collection.

With two collections, two results are produced per memory access. The time taken to generate one plane is the number

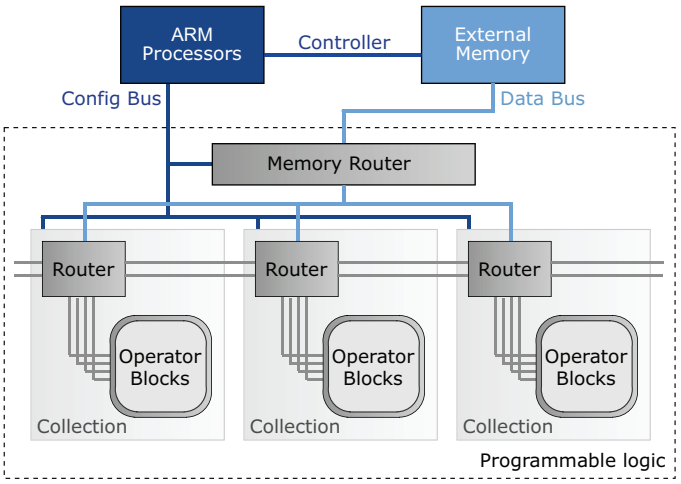
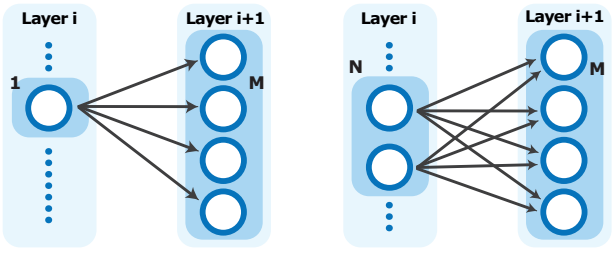


Fig. 1. Overview of the implemented system and data paths. The memory router is a crossbar switch. A local router in each collection is directly connected to routers in neighboring collections, thereby constructing a one dimensional torus-like data streaming network. The group of operator blocks in each collection contains convolution, max-pooling and non-linear operators.

of pixels in the input divided by the clock frequency of the collections plus an initial configuration delay. This delay is negligible compared to the size of the image. Such operation is possible due to the pipelined nature of the operators which produce one output per clock cycle and can be run in parallel. A router inside each collection routes incoming data from the memory router to the operators. This router also controls the flow of data from one operator to the next. Finally, it can also route two data streams to its east and west neighbors. The FPGA on the XC7Z020 chip holds up to two collections with a 10×10 convolution unit (convolver). We can fit more collections at the cost of having a smaller convolution unit (7×7 allows for four collections).

The convolution operation in DCNNs can be generalized as an N-to-M type as shown in figure 2. This results in two possible cases when running a feed-forward network. Our hardware design allows for efficient routing of data in both cases.



(a) 1-to-M convolution (b) N-to-M convolution

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

1) $N = 1$ case: The $N = 1$ case, as demonstrated in figure 2(a), occurs when there is one input stream and M filter kernels. This is typical of the first layer of a DCNN which takes in a greyscale image as its input. Such a layer would produce

M outputs and no intermediate results. In a 1-to-M convolution, resources are efficiently handled by minimizing the number of memory accesses and maximizing collection utilization. One input stream is routed to multiple collections by the memory router, as indicated by "data bus" traces in figure 1. It is then processed by the collection's operators in a pipelined fashion before being routed back to the memory router as one output of the current layer. The memory router then sends this output to memory where it awaits its turn to be sent back in as an input to the next layer. For this case, we need only $\lceil \frac{M}{C} \rceil$ memory accesses where M is the number of kernels and C is the number of collections. In a general purpose processor, this would result in $3 \times M$ memory accesses, one access for each operator.

Input images are generally large and cannot fit in first level caches of general purpose processors. Usually they get loaded into the larger last level cache. While access to these caches is much faster than main memory access, it is slower than accessing level one cache. This results in performance that is not significantly slower than peak performance but is not optimum either.

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

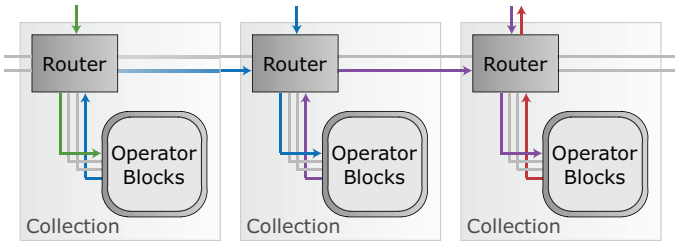


Fig. 3. Example of routing scheme for a 3-to-1 convolution. Inputs are streamed to each of routers. The input to each router is a stream coming from the memory router. The output of the operator blocks is the result of a convolution in the left-most collection. This is an intermediate result. It then gets combined by pixel-wise addition with the intermediate produced by the neighboring collection. This process occurs once more before the final output is produced. The stream flows along the path colored in green, blue, purple and red in that order. The connections between neighbors facilitate rapid data transfer between collections.

In our design, each collection has two ports to its east and west neighbors. These connections, illustrated in figure 3 enable a collection to send a stream to either of its neighbor without interrupting the memory router or the collection's operators. As long as a neighboring collection's operator is available, the intermediate result produced does not need to be sent back to memory either. It can be combined with another intermediate, effectively reducing memory accesses by a factor of two (two intermediates being transfered back and forth between memory and coprocessor).

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

3) Advantage over a grid: Compared to the processing grid structure described in [13], our design minimizes the number of global connections by grouping operator blocks but keeps the essential connections that are used for relaying streams in order to perform an N-to-M convolution. While having many connections to adjacent processing tiles gives flexibility in general processing, it does not necessarily benefit over the torus collection network (figure 1) when targeting DCNNs as their sequence of operations is the same, even in different network architectures. Such optimum routing lets the system overcome timing violation during hardware synthesis while also relaxing the complexitiy of scheduling operations. Since the torus structure casts the problem into a 1D search space as opposed to 2D space in a grid, much less effort is needed to find the best solution for the given deep network. For these reasons, the collection network is an optimum routing strategy for custom designed hardware for processing DCNNs.

V. Experimental Results

The performance of this system was compared to a system running an Intel Core i5 2.6GHz CPU, a NVIDIA GeForce GTX 690 GPU, and an ARM Cortex A9 processor. The experiments were conducted to demonstrate peak performance and performance in real-world applications. The applications used were a face detector and an object tracker [6] while the filter-bank demonstrates peak performance of this design. The number of operations in each network is listed in figures 4 and 5 to demonstrate the scale of each network. Performance per second and performance per watt are used as metrics for comparison. The results are reported in figure 4 and 5.

We used the Torch7 software [15] for demonstrating performance on different platforms. A 256×256 input image[1] was used in all applications. The benchmark aimed at maximizing the utilization on all platforms so as to record peak performance possible in each case and equal workload was assigned to each platform.

Figure 5 demonstrates power efficiency of the platforms. Our system records the highest performance per watt numbers in all applications. The GPU can exploit massive parallelism, but also consumes significantly more power than the other systems. Since parallelism is mostly obtained by the convolution operators, larger kernels and images give higher performance benefits. For this reason, highest performace was measured in the filter-bank for all platforms except Zynq ARM processor. Our accelerator was able to deliver 8.2 G-ops/s-W in the filter-bank demonstration that consists of 32 convolution with 10×10 kernels at the first layer.

Performance tends to drop for hidden layers with multiple inputs. Generally, using many convolutional planes or processing deeper layers causes more intermediates to be created

[1]Torch7 CUDA modules have a 256px limitation for width and height and are optimized to work in a batch mode.

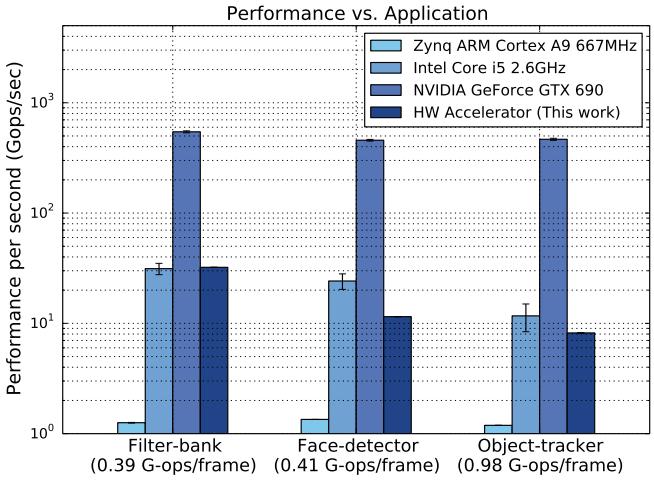


Fig. 4. Performance per second test over different networks in real-world applications. Tested on the accelerator with two collections each with 10×10 convolution kernels. A logarithmic scale is used on the y-axis.

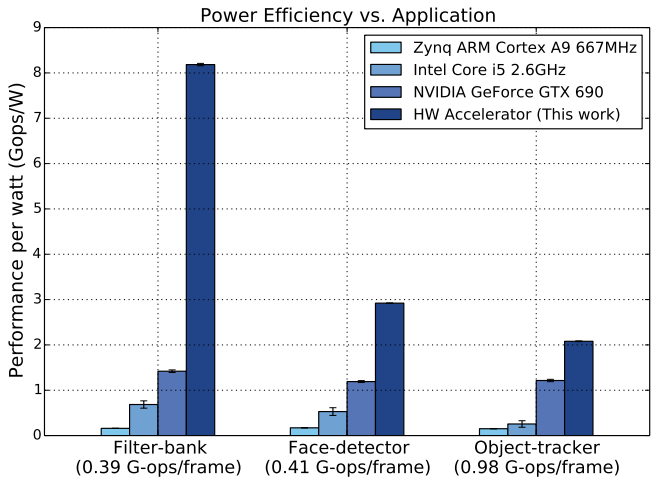


Fig. 5. Performance per watt test over different networks in real-world applications. Tested on the accelerator with two collections each with 10×10 convolution kernels. A linear scale is used on the y-axis.

during the entire computation. The face-detector and object-tracker used in this work contains two and three network layers, respectively. These applications have frequent memory access as compared to the filter-bank. Hence, this results in a performance drop which becomes significant for a deeper network. The power consumption when running the benchmark was measured to be 45.7 watts on the Intel CPU and 384 watts on the GPU whereas our entire platform consumed a maximum of 3.9 watts.

VI. Conclusion

We present an efficient implementation of a hardware accelerator for deep convolutional neural networks. Our accelerator is a scalable mobile platform that consumes less than $4\,\mathrm{W}$ and gives a peak performance of $40\,\mathrm{G}$-ops/sec. We describe a routing strategy that enables us to efficiently utilize hardware resources to obtain high performance in real-world applications. The platform's portability combined with the efficient routing strategy and low power make it feasible to use this design in applications that require a mobile coprocessor capable of running DCNNs in real-time.

Acknowledgment

This work is supported by 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)," *Computer Vision and Image Understanding*, 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," in *Advances in Neural Information Processing Systems*, vol. 25, 2012.

[5] R. Socher, B. Huval, B. Bath, C. D. Manning, and A. Ng, "Convolutional-recursive deep learning for 3d object classification," in *Advances in Neural Information Processing Systems*, 2012, pp. 665–673.

[6] 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.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *Deep Learning, Neural Information Processing Systems Workshop*, 2013.

[8] 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.

[9] 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, 2010, pp. 273–284.

[10] 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*, 2009, pp. 529–536.

[11] U. Kapasi, S. Rixner, W. Dally, B. Khailany, J. Ahn, P. Mattson, and J. Owens, "Programmable stream processors," *Computer*, vol. 36, no. 8, pp. 54–62, Aug 2003.

[12] 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.

[13] 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.

[14] M. Peemen, A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, Oct 2013, pp. 13–19.

[15] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, Neural Information Processing Systems Workshop*, 2011.