# Memory Access Optimized Routing Scheme for Deep Networks on a Mobile Coprocessor

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

***Abstract*—In this paper, we present a memory access optimized routing scheme for a hardware accelerated real-time implementation of deep convolutional neural networks (DCNNs) on a mobile platform. DCNNs consist of multiple layers of 3D convolutions, each comprising between tens and hundreds of filters and they generate the most expensive operations in DCNNs. Systems that run DCNNs need to pass 3D input maps to the hardware accelerators for convolutions and they face the limitation of streaming data in and out of the hardware accelerator. The bandwidth limited systems require data reuse to utilize computational resources efficiently. We propose a new routing scheme for 3D convolutions by taking advantage of the characteristic of DCNNs to fully utilize all the resources in the hardware accelerator. This routing scheme is implemented on the Xilinx Zynq-7000 All Programmable SoC. The system fully explores weight level and node level parallelization of DCNNs and achieves a peak performance 2x better than the previous routing scheme while running DCNNs.**

## I. Introduction

Artificial vision systems aim to provide visual understanding by extracting high-level information from raw images. In other words, these systems aim to process high dimensional data like images and videos and extract useful low-dimensional data, where decisions can be made based on. The exploration of such systems has been an active field of research for the past decades and many recent algorithms are showing promise for use in visual understanding. 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 (HMAX) [3]. Recent work on DCNNs show great promise to help solve visual classification problems [4]–[7].

The power of DCNNs comes from having many layers and filters to extract features from images in a hierarchical manner. They consist of multiple layers of convolutions, each comprising between tens and hundreds of filters. Other than convolutions, each layer also includes a pooling and a non-linearity operation, as shown in Figure 1. The first convolution layer extracts simple features like edges and corners and pooling provides scale and distortion invariance to the network. The second convolution layer, because it is extracting features from the output maps of the first layer, extracts more complex shapes. As the network goes deeper, the higher layers extract more complex shapes and with the pooling after each convolution, they become less invariant to distortion in images.

DCNNs are getting bigger with more layers and parameters as new methods prevent them from over-fitting [8] and Graphics Processing Units (GPUs) provide fast training of DCNNs. As DCNNs get bigger, the accuracy of object detection increases [4], which makes them useful for applications in autonomous robots, security systems, micro-UAVs and more recently, mobile phones, automobiles and wearable support systems [9]. These application require algorithms that can recognize objects with a high degree of accuracy, however, they should also be executed in real-time which require a custom hardware. This is especially because convolutions are computationally very expensive [10].

GPUs are becoming a common alternative to custom hardware in vision applications because they are inexpensive and easily programmable [11]. However, custom hardwares have better performance with less power consumption which is a must especially for mobile platforms. By developing a custom architecture and configuration library that are fully adapted to DCNNs, the product of power consumption by performance can be improved by two orders of magnitude (100x). Because of these advantages, extensive research has been done on the custom architectures for convolutional networks or similar algorithms [12]–[15].

This paper presents a memory access optimized routing scheme for a hardware accelerated real-time implementation of DCNNs on a mobile platform. Operations are scheduled by taking advantage of the characteristic of DCNNs which enables data reuse for the full utilization of the hardware resources.

The rest of the paper is organized as follows: Section II explains related work. Section III describes the architecture of the custom hardware. The capabilities and limitations of the custom hardware are given in this section which are the most important considerations when we optimize the memory accesses. Section IV describes the resource allocation that is optimized for DCNNs. This section describes the main contribution of this work. Section V gives results on the performance of the system. Section VI discusses the results and talks about the future work. Section VII concludes.

## II. Related Work

Due to the intense and massive computation of deep convolutional neural networks (DCNNs), several types of parallelism has been applied to achieve real-time processing such as weight parallelism which corresponds to a parallel sum of products computation in convolution and node parallelism which corresponds to computation over multiple convolutional planes [16].

Extensive research has been done to exploit these parallelisms of DCNNs on CPUs, GPUs and custom hardware. A straight forward implementation of convolution operations contains multiple nested for loops. For CPUs, unrolling the

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

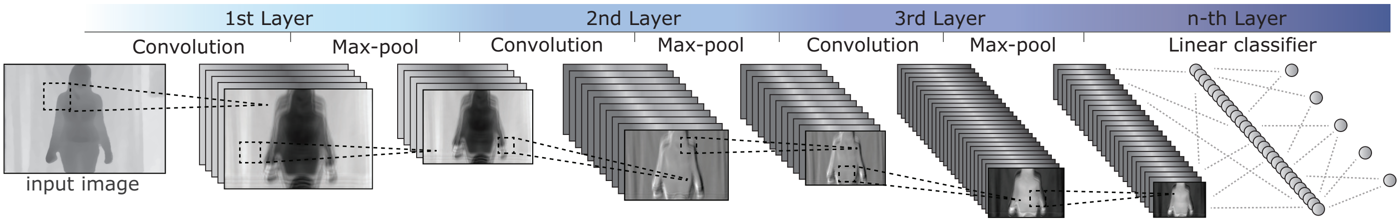


Fig. 1. Diagram of a DCNN. Each layer consists of three consecutive operations - convolution, max-pooling and non-linearity. Non-linearity is not demonstrated in this figure as it does not change the dimension of the outputs. Based on the number of filters at each convolution layer, the number of planes changes and based on the max-pooling stride the width and height of the planar maps decrease. The last layer is a spatial classifier which translates the set of inputs onto a set of outputs which can be the labels of objects contained in input image.

convolution operations for cache friendly memory access is suggested by the paper [17]. Other optimized software libraries are implemented [18], [19]. Despite the benefit of such implementation, DCNNs on general purpose processors are still too demanding to be used for real-time applications.

Different communication optimizations for GPUs are explored by the papers [20]–[22] to overcome the bandwidth limitation between the host system and GPU memory. The challenge of GPUs are to use the small local memory spaces efficiently which only can store a part of an image. However, the problem with the GPUs is the limited cache memory for storing the filter coefficients of large networks [23].

Many research groups explored custom hardwares to accelerate convolutional neural networks [24]–[27]. Custom hardwares can exploit massive parallelism compared to CPUs while consuming low power unlike GPUs because typically a large number of logic units that can operate in parallel is specialized to operations required for DCNNs. Especially node parallelism gives a significant performance benefit for convolutional neural networks [16], [24], [27] based on the fact that each convolutional plane is independent from others in the same layer. The degree of parallelism can be increased by placing as many processing units as a silicon can hold.

However, while such powerful computing power achieved by parallel processing units provides high throughput, its actual performance is often limited by its memory bandwidth during DCNNs 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 which needs to be stored in memory [28]. Such intermediates require frequent memory access between host processor and memory. This causes significant overhead time when used for large-scale neural networks.

For this reason, efficient routing scheme as well as massive parallelism is a crucial factor for DCNNs accelerator in order to reduce data transfer overhead. This issue was demonstrated in a neuron spiking model [29] and a convolutional neural networks model [27]. However, no experimental result was supported to demonstrate performance of the former in a real-world scenario. The latter focus on smaller convolution kernels which do not have as much parallelism as larger kernels and may not be much faster on custom hardware than on general purpose processors.

The system described in this work uses a shared memory architecture where the same memory can be accessed by both ARM processor and programmable logic. The problem with this system is the port limitation to steam data in and out of the hardware accelerator. We propose a memory access optimization routing to overcome this limitation which enables the maximum node level parallelization of DCNNs. Whereas the hardware fully explores the weight parallelism, our routing scheme enables the full node parallelism with available resources in the hardware accelerator.

## III. Hardware Accelerator

In this section, we present the hardware accelerator for DCNNs as the capabilities and limitations of this hardware were the most important considerations when we optimized the routing scheme to decrease the number of memory accesses and increase the utilization of the hardware.

The hardware system is implemented on the Xilinx Zynq-7000 All Programmable SoC which has a shared memory architecture that has synchronous access to memory through both ARM processor and programmable logic. The system has four high performance ports to DDR3 memory. The high performance ports tap into DDR3 memory using the AXI4 bus. Each AXI DMA is bidirectional and can transfer two 32-bit data words per clock cycle.

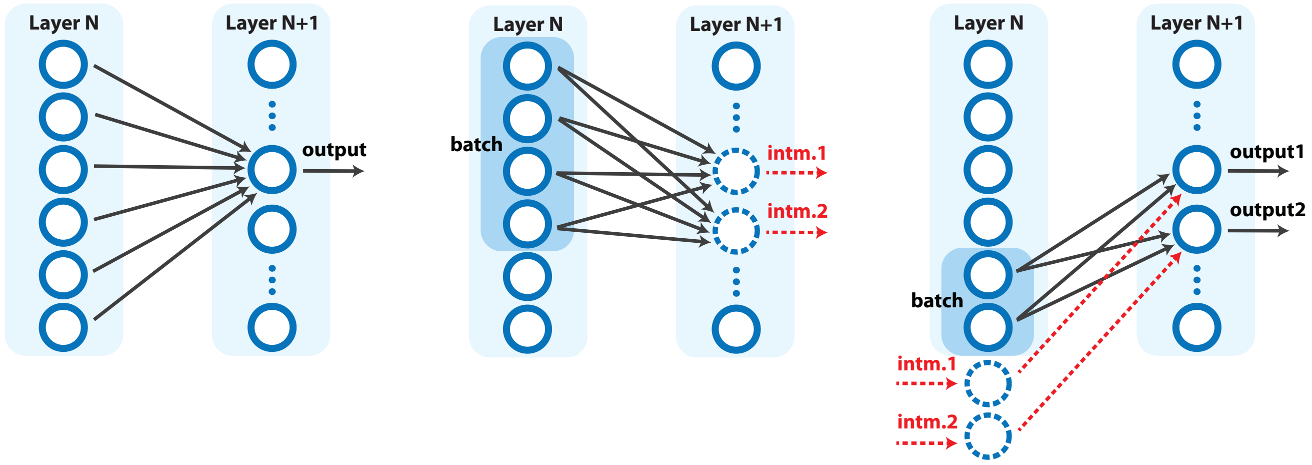
### A. Computational Resources

The operations that are commonly used in DCNNs are implemented in the custom hardware. We have the following computational resources in the custom hardware:

*1) Convolver:* The core of DCNNs, as the name suggests, is convolution operation. Convolution with trained filters are used to extract useful features from the input images or from the output maps of the previous layers, in which case convolution extracts complex features. This system's convolution engine can perform the following operation in one clock cycle.

$$y_{ij} = \sum_{m=1}^{n} \sum_{l=1}^{n} x_{i+m,j+l} w_{ml} \tag{1}$$

where, $y_{ij}$ is the output, $x_{ij}$ is the value of an input pixel and $w_{ml}$ is the value of the $n \times n$ filter kernel. As an image is streamed in, one input pixel results in one output pixel which provides full exploration of weight-level parallelization, not including an initial set up delay that occurs due to the pipelined nature of the hardware.



(a) Previous implementation

(b) Our implementation - step 1

(c) Our implementation - step 2

Fig. 2. Routing schemes for 3D convolutions. Solid circles represent the output maps of the corresponding layers. The ones on the left are the output maps of Layer N, so they are the input maps of Layer N+1. Each map is two dimensional. The arrows represent 2D convolutions and the arrows going to the same output are summed together. The libraries for the CPUs implement the convolution operation as in the Figure 2(a) where all the operations to produce one output are scheduled first. After, the calculations to produce one output are completed, the operations for the next outputs are scheduled. CPUs have access to the input maps, this scheme is an efficient way of implementation. However, a custom hardware has a limited access to the input maps. Therefore, the efficient way of implementation, 2(b) and 2(c), is to do the calculations together if they use the same inputs. The dashed circles represent intermediate values, which are the partial summation of the outputs of 2D convolutions. They need further processing for the final output.

*2) Max-pooler:* Max-pooling is a winner-take-all operation, the maximum activated value is transferred to the output while the other values from that local area are ignored. This operation gives roboustness to the networks because it discards the exact position of the extracted features. The max-pooler returns an image that is subsampled of the image in both dimensions.

*3) Non-linear operator:* DCNNs generally have a non-linear operation following the pooling operation. They are used to remove the unimportant information and enhance the important ones for the subsequent layers. Rectified Linear Unit (ReLU), $f(x) = \max(0; x)$ currently is one of the most widely used non-linear operator [4]. This non-linear operator in the custom hardware also produces one output per clock cycle.

*4) Stream Adder:* This module takes two streams and adds them together. This module is needed for 3D convolutions. Multiple 2D convolutions are calculated by the convolution engine and summed up to calculate 3D convolution. This operator produces one output per clock cycle.

### B. Architecture

The hardware is divided into two main areas: the operators required for processing images and the memory router.

*1) Collection:* The operators are bundled together into a single module called a collection. The presented system can hold eight collections. Each collection can be run in parallel, as can the operators contained within them. A collection has a convolution, a max-pooling, a non-linear modules and a stream adder. 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-linear module in the same collection.

Another important optimization of this hardware accelerator is the interior routing of the collections. Each collection has a port to its neighbor collections which is a big advantage for 3D convolutions because 3D convolutions require summation of multiple 2D convolved results. The convolver in each collection can perform 2D convolutions and produce intermediate results. The intermediate results can be sent to the neighbor collection's stream adder and be combined with the intermediate result produced in that collection. This process can be repeated among all collections. Therefore, if we have enough bandwidth to stream eight input maps, with eight collections we can calculate eight, 2D convolutions and sum all of them together at one DMA transaction which effectively reduces the memory access.

*2) Memory Router:* The memory router interfaces with the four AXI DMA engines and can be configured to route the incoming data streams to one or more outputs. Therefore, same incoming streams can be fed into different collections by the memory router.

### C. Limitations

This system can hold eight collections. However, there are only four AXI ports to stream data in and out. Therefore, the biggest handicap of this system is streaming data in and out of the hardware accelerator. We cannot stream different data to each collection. We need to use the same data in multiple collections for the full utilization. When we designed the routing scheme, this issue was the main concern.

## IV. Optimal Routing Scheme

The novelty of this work is the routing scheme described in this section. First, we want to introduce the most common and heavy operation in DCNNs. Generally, DCNNs take RGB images as inputs, these images have dimensions of $3 \times \text{Width} \times \text{Height}$ and in subsequent layers the first dimension increases based on the number of filters at each layer. For example, if

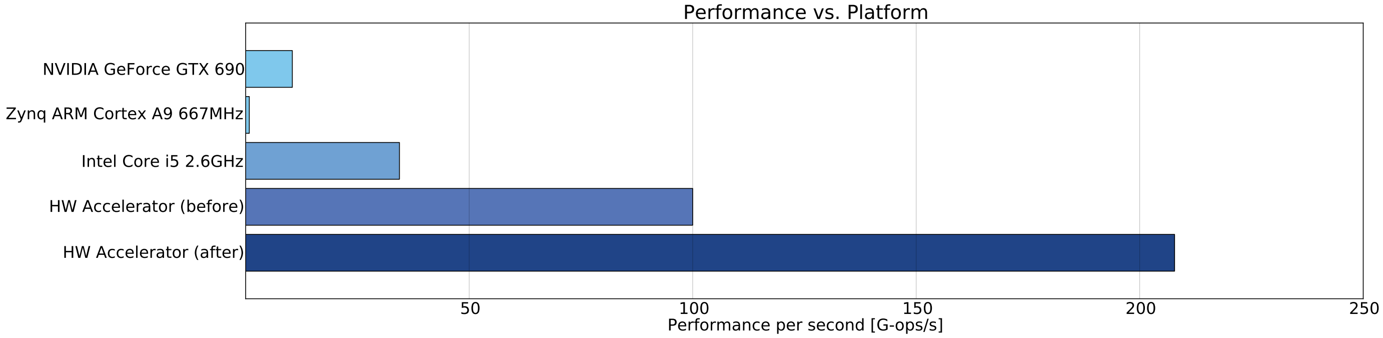


Fig. 3. Performance per second test over one layer network on different platforms. The network consists of $4 \times 32$ convolution operations with $10 \times 10$ filters, $2 \times 2$ max-pooling and non-linear operation. We also compare the hardware accelerator that is configured with the previous implementation described above (before) and our implementation (after).

the first layer has eight filters, the input of the second layer's first dimension becomes eight. State of the art DCNNs contain hundreds of filters at each layer.

Therefore, the filters for convolution are three dimensional and so convolution operation includes a third dimensional addition for each output as in equation 2.

$$y_{ij} = \sum_{k=1}^{n} \sum_{m=1}^{n} \sum_{l=1}^{n} x_{k,i+m,j+l} w_{kml} \tag{2}$$

This operation is also demonstrated in Figure 2, each line refers to a 2D convolution operation and the arrows going to the same output are summed together.

In our system, as stated before, equation 1 can be performed in one clock cycle and each collection can process a 2D input image by convolving it with 2D filters. In DCNNs, we need to process images with 3D filters. Therefore, we need to sum all input images convolved by 2D filters to obtain one output map. To produce one output map, we need to convolve N input images with $N \times W \times H$ filters which can be calculated as N separate 2D convolutions and their summation.

This hardware accelerator can perform 2D convolutions at each collection and the results can be streamed into the neighbor collection's stream adder for summation. Usually, there are not enough resources to calculate all 2D convolutions needed to produce one output map. Hence, the output of each DMA transaction is saved as an intermediate value. Here, intermediate value refers to the partial summation over $k$ in equation 2. In the next transaction, the intermediate values are streamed into the custom hardware for further accumulation with the other outputs from 2D convolution operations.

In this custom hardware, we have four ports to transfer inputs and outputs between the eight collections and memory. Therefore, we do not have enough ports to bring a different input to each collection. However, we can send four different inputs to the first four collections and transfer the same four inputs to the other four collections. As DCNNs process the same inputs with different filters, we take advantage of this input sharing which is illustrated in Figure 2(b) and Figure 2(c).

The straightforward approach, also used in the libraries for the CPUs, is to process the operations to calculate one output map, as shown in the Figure 2(a). In this scheme, the routing is based on the outputs; outputs are calculated one after the other. CPUs have easy access to the inputs, therefore, this scheme is an efficient way of implementation. However, custom hardware has limited access to the input maps. Because of that, our scheme is based on the inputs. Once we stream an input, we want to do operations together that are using that input so that we can avoid streaming the same input multiple times.

With the approach from Figure 2(a) because we have four ports to bring data, we can only utilize four collections. However, we know that to calculate the other outputs, we will process the same inputs with different filters. Therefore, to utilize all eight collections, when we bring four inputs in, we begin calculating intermediate values that will produce two outputs later as depicted in Figure 2(b). In the next step, Figure 2(c), we keep performing the necessary convolutions and also sum the output of the convolutions with the intermediate value from the previous cycle.

Therefore, when we allocate resources, we use half of the collections for the operations that will produce one of the output maps and the rest for the next output map. The operations for the two subsequent output maps are scheduled together. When an input streams in from one port, it allocates two collections because of the resource sharing described above. In DCNNs, for each output map, we generally have tens and hundreds of input maps. Therefore, by this scheme we can produce two intermediate maps after each DMA transaction and two outputs after the required number of transactions. Because there are four ports to stream data out, streaming output is not limited by the bandwidth.

Every time we calculate the final output of the 3D convolutions, we stream the output to the max-pooler and nonlinear operators in the same collection. Therefore, the final output of the layer is streamed to memory. This routing scheme decreases the memory access 2x and increases the hardware utilization 2x compare to the original scheme.

## V. Experimental Results

The performance of the system that is configured with the memory access optimized routing scheme was compared to the system that is configured with the previous routing and also a system running an Intel Core i5 2.6GHz CPU, a NVIDIA GeForce GTX 690 GPU, and an ARM Cortex A9 processor.

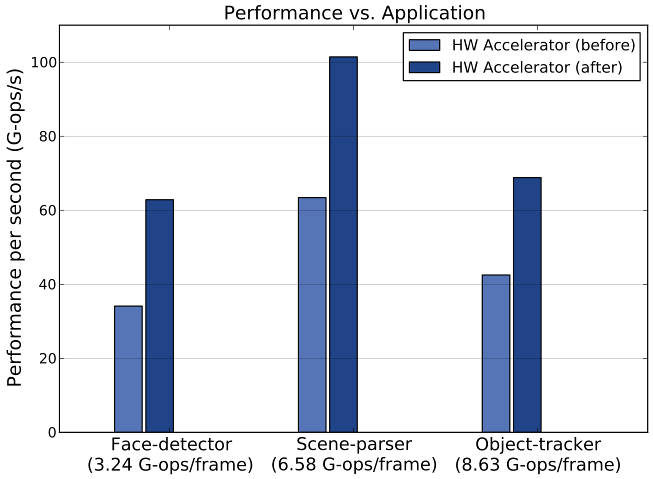


Fig. 4. Performance per second comparison of different applications of DCNNs between the previous routing of DCNNs (before) and this work (after).

We used the Torch7 software [19] for demonstrating performance on different platforms. The experiments use an input image of $4 \times 256 \times 256$. [1]

The experiments are designed to demonstrate DCNN applications for video or image processing. In the applications, the images need to be processed one by one which results in the poor performance of GPU. GPUs can process multiple images at the same time and Torch7 CUDA modules are optimized to work in a batch mode. Therefore, as the batch size (number of input images) are increased the performance increases linearly. Our experiments show that with one input image, the GPU performs $10.43$ G-ops/s, if we increase the number of images to 128, the performance increases to $500$ G-ops/s.

The DCNN implementation of GPUs are mostly used to train the networks where you have abundant number of input images. GPUs have not been explored extensively to run real time DCNN applications. An important reason for that is the high-power consumption of GPUs. GPUs consume $384$ watts, whereas CPUs consume $45.7$ watts and our platform consumes a maximum of $8$ watts.

We also compare the hardware accelerator that is configured with the previous routing scheme which is the straightforward implementation that is mostly used libraries for CPUs described above (before) and our implementation (after). In this test, we choose the input image as $4 \times 256 \times 256$ which does not produce intermediate results and provide the full utilization of our implementation. While our implementation is able to utilize all the resources and reports $207.8$ G-ops/s, the previous implementation utilizes half of the resources because of the limited bandwidth and gives us $100$ G-ops/s. The hardware accelerator with the memory access optimized routing scheduling is $270$ times faster than the baseline application processor, a dual core ARM Cortex A9.

We also tested the performance in real-world applications as reported in the Figure 4. We trained DCNNs according to the method described in the paper [30] for face detection, scene parsing and object tracking. DCNN for face detection and scene parsing contain two layer networks and object tracking contains three layer networks. The RGB input images with the dimension of $3 \times 500 \times 500$ are fed into the custom hardware. Each network for each application requires different number of G-ops per frame as shown in the Figure 4.

We compared the performance of the previous routing with the memory access optimized routing proposed in this work. Approximately, this work increased the performance by 2x in all three real-world applications we implemented.

## VI. Discussion

In this section we analyze the reported performance of the proposed system in the Section V. The system achieved different number of G-ops/s at each application because of the different amount of possible parallelizations. First application face-detector uses small filter sizes which makes the system explore small number of weight-level parallelizations.

Also as the network layers increase, there is a performance drop because of the frequent memory accesses compared to the one layer network from the Figure 3. The suggested memory access optimized routing scheme decreases the memory accesses 2x compared to the previous implementation.

Because of the specifications of the current system, this optimization scheme is implemented as calculation of two outputs synchronously. However, this routing scheme can be optimized for different platforms with different number of ports and computational resources. If the system has more computational resources than the system presented here, three or more outputs can be calculated at the same time.

This work does not apply to the spatial classifier. Future work will be the exploration of the parallelizations in the classifier module.

## VII. Conclusions

We describe a memory access optimized routing scheme for a hardware accelerated real-time implementation of DCNNs on a mobile platform, and a complete system that can run real-time DCNN applications.

The focus of this work was reusing the inputs for 3D convolutions. DCNNs consist of multiple layers of 3D convolutions, each comprising between tens and hundreds of filters. Systems that run DCNNs need to pass 3D input maps to the hardware accelerators for convolutions and they face the limitation of streaming data in and out of the hardware accelerator. As the biggest limitation is the bandwidth, and the same input maps are processed with multiple different filters, the operations are scheduled to calculate partial multiple outputs instead of one output. This optimization enables to use all hardware resources despite of the limited bandwidth.

The system fully explores weight-level and node-level parallelization of DCNNs and achieves a peak performance of 210 G-ops/s while running DCNN whereas the previous routing achieves 100 G-ops/s.

## Acknowledgment

Work supported by Office of Naval Research (ONR) grants 14PR02106-01 P00004 and MURI N000141010278.

[1] Torch7 CUDA modules have a 256px limitation for width and height.

# 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] 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, Aug 2013.

[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] 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.

[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. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1207.html#abs-1207-0580

[9] T. Starner, “Project glass: An extension of the self,” *Pervasive Computing, IEEE*, vol. 12, no. 2, pp. 14–16, 2013.

[10] 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.

[11] A. Coates, P. Baumstarck, Q. Le, and A. Y. Ng, “Scalable learning for object detection with gpu hardware,” in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*. IEEE, 2009, pp. 4287–4293.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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, to be published.

[29] S. Moradi, N. Imam, R. Manohar, and G. Indiveri, “A memory-efficient routing method for large-scale spiking neural networks,” in *Circuit Theory and Design (ECCTD), 2013 European Conference on*, Sept 2013, pp. 1–4.

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.