Efficient FPGA Acceleration of Convolutional Neural Networks Using Logical-3D Compute Array

Atul Rahman and Jongeun Lee*
UNIST (Ulsan National Institute of Science and Technology)
Ulsan, Korea

Kiyoung Choi
Seoul National University
Seoul, Korea

***Abstract*—Convolutional Deep Neural Networks (DNNs) are reported to show outstanding recognition performance in many image-related machine learning tasks. DNNs have a very high computational requirement, making accelerators a very attractive option. These DNNs have many convolutional layers with different parameters in terms of input/output/kernel sizes as well as input stride. Design constraints usually require a single design for all layers of a given DNN. Thus a key challenge is how to design a common architecture that can perform well for all convolutional layers of a DNN, which can be quite diverse and complex. In this paper we present a flexible yet highly efficient 3D neuron array architecture that is a natural fit for convolutional layers. We also present our technique to optimize its parameters including on-chip buffer sizes for a given set of resource constraint for modern FPGAs. Our experimental results targeting a Virtex-7 FPGA demonstrate that our proposed technique can generate DNN accelerators that can outperform the state-of-the-art solutions, by 22% for 32-bit floating-point MAC implementations, and are far more scalable in terms of compute resources and DNN size.**

I. Introduction

As Deep Neural Networks (DNNs) are repeatedly shown to outperform other algorithms in several machine learning problems, an important question today is how to accelerate the copious amount of computation for DNNs. In particular convolutional DNNs, which are inspired by visual cortex and are good at image understanding tasks [1], expands the connectivity of a neuron to three dimensions, raising computational requirement dramatically compared with traditional types of neural networks.

While GP-GPUs are currently the most commonly used platform in accelerating DNNs especially when training is involved [2], FPGAs can provide higher energy-efficiency, which has a great appeal for portable and embedded DNN applications.

Previous work on FPGA acceleration of convolutional DNNs [3], [4], [5] shows progressively increasing throughput. FPGA based designs can either be computation or memory bandwidth bound. The work in [6] only considers optimizing FPGA based CNN design reducing the memory bandwidth requirement. However, for computation bound designs the proposed accelerator is not optimized. The parallelism scheme of [7] and [4] is same and the data reuse factor is very low due to not effective use of on-chip buffers. The approach in [3] was to accelerate CNN with both software and hardware implementation. The implementation of the processor was ad-hoc and not generalized design methodology for any given CNN topology and FPGA platform specific constraints. The most recent one [8] takes a high-level synthesis approach, and by optimizing for both computation rate and communication bandwidth, it can achieve 61.62 GFLOPS on a Virtex-7 FPGA when applied to a real-life DNN [1] with five convolutional layers, using 5 DSP units per multiply-add operation.

In this paper we present a highly optimized DNN accelerator architecture that can push the performance envelope even further, in terms of both computation and communication.

One key challenge in raising performance is how to design a common architecture that can perform well for *all* layers, overcoming the diversity and complexity of computations across layers (see Table I). Fully parallel hardware implementation is simply impossible due to the sheer number of operations needed, making it crucial to choose the best parallelization scheme.

Our solution is a novel architecture called *Input-recycling Convolutional Array of Neurons (ICAN)*, which differs from previous solutions by parallelizing along three carefully chosen dimensions that mimics the convolution operation used in DNNs. We show through empirical analyses that structuring compute elements in a 3D rather than 2D topology can significantly improve the utilization of key FPGA resources such as DSP slices for various convolutional layer shapes. A potential drawback of our ICAN architecture, however, is its internal complexity. We address this problem with our *Input Reuse Network*, which can minimize routing complexity and exploit data reuse opportunity inherent in convolution operations.

Another key factor contributing to the high efficiency of our solution is its very aggressive parameter optimization considering *memory* as well as *compute* resources, which can greatly improve the computation-to-communication ratio of our accelerators.

When applied to a real-life DNN [1] our ICAN accelerator can not only achieve 22% higher performance on a Virtex-7 FPGA compared with the state-of-the-art solution [8], but the designs optimized by our algorithm are far more scalable than those of the previous work in terms of compute resources and DNN size.

The rest of the paper is organized as follows. In Section II

This work was supported in part by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1501-08.

J. Lee is the corresponding author of this paper (E-mail: jlee@unist.ac.kr).

TABLE I
DIVERSITY AND COMPLEXITY OF CONVOLUTIONAL LAYERS OF A REAL-LIFE DNN [1] (SHOWN ONLY FOR ONE OF THE TWO PARTITIONS)

Layer	#Input neurons (Z, Y, X)	#Output neurons (M, R, C)	Misc. K	S	Complexity #MAC Op.*
1	$(3, 224, 224)$	$(48, 55, 55)$	11	4	105M
2	$(48, 27, 27)$	$(128, 27, 27)$	5	1	224M
3	$(256, 13, 13)$	$(192, 13, 13)$	3	1	150M
4	$(192, 13, 13)$	$(192, 13, 13)$	3	1	112M
5	$(192, 13, 13)$	$(128, 13, 13)$	3	1	75M

The number of MAC operations (#MAC Op.*) is for both partitions combined.

we first consider the problem of maximizing the throughput of *any* accelerator that can be implemented on a modern FPGA, and propose our parallelization scheme that is flexible yet highly efficient for convolutional layer computations. Section III presents some of the details of our accelerator architecture along with our aggressive optimization strategy. We present our experimental results in Section IV and conclude the paper in Section V.

II. MAXIMIZING COMPUTATION RATE

In this section we consider the computation of only one convolutional layer; one layer is usually long enough to ameliorate the switching overhead between layers if the switching does not involve FPGA reconfiguration, which is the case with our accelerator. Also we assume that on-chip buffers are large enough to avoid any memory bottleneck; the issue of limited on-chip buffers is addressed in Section III-C.

A. DSP Utilization

Maximizing the computation rate on today's FPGAs often boils down to maximizing the utilization of DSP slices such as DSP48E1 in Xilinx Virtex-7 FPGA, since DSP slices are far more efficient than any other implementation on an FPGA for common arithmetic operations. For instance, the mentioned DSP slice can perform a 25x18-bit multiplication followed by a 48-bit addition in just one cycle.

A convolutional layer is essentially a set of many multiply-accumulate (MAC) operations, and is thus a perfect fit for DSP slices. Obviously, increasing the DSP slice utilization, which we simply refer to as *DSP utilization*, is one effective way to increase the throughput of a system.

For any DNN accelerator implementing MAC operations using DSP slices only, DSP utilization can be calculated as:

$$\textit{DSP utilization} = \frac{\textit{\#MAC units} \times \textit{\#DSPs per MAC unit}}{\textit{Total \# of available DSPs}} \quad (1)$$

Here *#MAC units* is the number of MAC units in the DNN accelerator, which is a design parameter, and *#DSPs per MAC unit* is determined mainly by the precision of the MAC operation. Thus given an FPGA and the precision of a MAC operation, we can determine the maximum number of MAC units that can be included in an accelerator.

B. MAC Utilization

The other factor of the computation rate is *MAC utilization*, short for the utilization of MAC units of a DNN accelerator. The key observation is that not every MAC unit in a design is utilized at all times, and that this utilization varies depending on how we map a DNN to the accelerator. We can compute MAC utilization if we know how many cycles it takes for a DNN accelerator to complete a convolutional layer, which is denoted by *#exec. cycles* in the following equation. For brevity we assume that MAC units have the throughput of one.

$$\textit{MAC utilization} = \frac{\textit{Total \# of MAC ops in DNN}}{\textit{\#MAC units} \times \textit{\#exec. cycles}} \quad (2)$$

Then the computation rate is proportional to the product of the two utilization factors, and determines the system throughput if input and output are always ready; otherwise, any wait cycles for input/output need to be added to the total execution time.

While DSP utilization is commonly reported by a synthesis tool and usually considered in previous work as well, consideration of MAC utilization is new. We find that MAC utilization of the previous work [8] sometimes quite low, suggesting a scope for improvement. For further analysis of MAC utilization, we need to look into the structure of DNN computation.

C. Structure of Convolutional Layer Computation

Computationally a convolutional layer is a mere transformation of a 3D array into another 3D array using a series of 2D convolutions extended into the third dimension. Therefore computing one output point requires a 3-deep nested loop, since the output is arranged in a 3D array we need a 6-deep nested loop to complete the computation of one convolutional layer. The body of the loop nest is a simple MAC operation just like in matrix multiplication, and all the loop levels are permutable if we handle a bias term as an extra term in convolution.

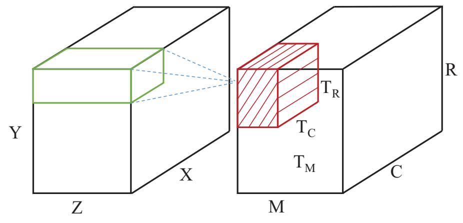
A key question in mapping this loop nest is which loops to *parallelize* vs. which ones to *iterate*. *Parallelization* means replicating the hardware resources, thus requiring more DSP units, whereas *iteration* means the particular loop level is done sequentially, thus more execution cycles. Obviously parallelization is limited by the maximum number of MAC units derived from (1), but depending on how exactly we do it, MAC utilization may vary. Thus the goal of our accelerator design is to maximize MAC utilization for various layer shapes while not incurring high overhead in terms of implementation (e.g., routing) or on-chip buffer size requirement.

D. Our Proposed Parallelization Scheme

Figure 1(a), which is a tiled and HW-unrolled version of a generic 6-deep convolutional layer code, illustrates our proposed parallelization scheme, including the selection of HW-unrolled loops (*which loops to parallelize*) and the order of loops (*in which order to iterate*). In the code, A, B, W_m are input, output, and weights, respectively.

```
for (mm=0; mm<M; mm+=T_M)
 for (z=0 ; z <Z; z++)
  for (rr=0; rr <R; rr +=T_R)
   for (cc=0; cc<C; cc+=T_C)
    for (y=0; y<K; y++)
     for (x=0; x<K; x++)                      HW UNROLL
       for (m=mm; m<min(mm+T_M,M); m++)
        for (r=rr; r<min(rr+T_R,R); r++)
         for (c=cc; c<min(cc+T_C,C); c++)
           B[m][r][c] += W_m[z][y][x] *
                         A[z][S*r+y][S*c +x];
```

(a) Our parallelization scheme in C code



(b) Our parallelization scheme

Params	Description
Z	# of input feature maps
Y, X	Input height and width
M	# of output feature maps
R, C	Output height and width
K	Kernel size in one dimension
S	Stride in the input

(c) Convoluational layer parameters

Fig. 1. Our proposed parallelization scheme.

Though the code might seem like a simple application of known loop transformations such as loop tiling, loop interchange, and loop unrolling, finding an optimal transformation can be elusive due to the large number of combinations and the difficulty of quantitative evaluation with symbolic variables.

The HW-unrolled iterations are implemented as parallel compute elements, which can be called *compute tile*. Compute tile is shown in red (shaded) in Figure 1(b).

The unique features of our solution include the following. First, our compute tile is 3D and therefore there are more tile parameters than in a 2D compute tile, which often gives higher MAC utilization simply because there are more parameter combinations to try out. Second, the three dimensions of our compute tile are exactly the same as the output array dimensions. In other words, each MAC unit corresponds to one neuron (for enough tile sizes), which seems like a natural and straightforward implementation. By contrast, in the parallelization scheme of [8], one MAC unit corresponds to entire neurons in the RC plane of the output, creating artificial time-sharing of resources. Third, our compute tile is still flexible enough to support various shape parameters (those listed in Figure 1(c)) as demonstrated in our experiments. Fourth, our compute tile boasts high reuse factor for input data (temporal reuse for m-loop, spatial reuse for r- and c-loops) and weight parameters (shared by all neurons in the RC plane). Finally, the on-chip buffer size requirement of our solution to realize maximal data reuse is not high. Compared with [8] our buffer size requirement for *maximal data reuse* is lower for weight/input buffers, and higher by M/T_M for output buffer only.

III. Architecture and Optimization

A. Architecture Overview

Figure 2 illustrates our DNN accelerator architecture. It implements one convolutional layer computation without including max-pooling or activation functions. The same datapath is used for every convolutional layer of a DNN, and the specifics of each layer such as different input/output/kernel sizes are taken care of by hardware controller. All data are stored in the external DRAM, including input/output feature maps and weight parameters. For faster processing, the computation

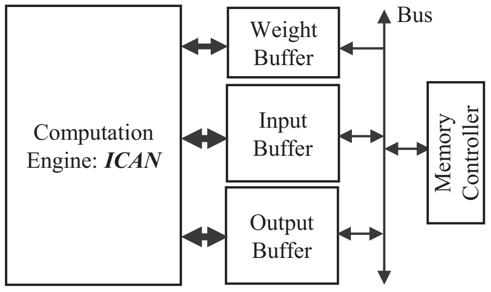


Fig. 2. The top-level view of our accelerator architecture, which does one layer computation at a time. All three buffers are double-buffered.

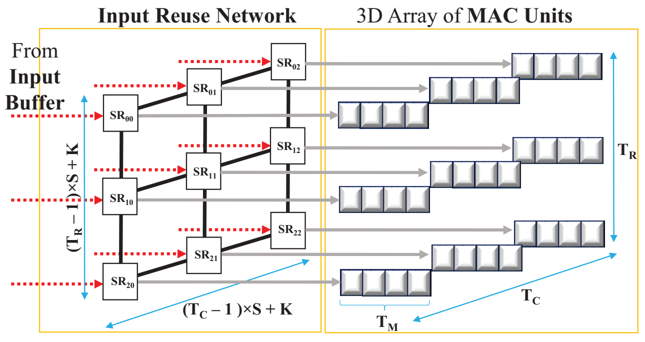


Fig. 3. Main components of ICAN: input reuse network and compute tile. Compute tile, which is a 3D array of $T_M T_R T_C$ MAC units, takes additional input from the weight buffer, and its output is stored in the output buffer.

engine, *ICAN*, uses 3 on-chip buffers, *input*, *output*, and *weight*, which are all double-buffered to hide external memory access latency. In addition to what is shown, there is a small processor for miscellaneous tasks such as host interface and memory initialization.

B. ICAN: Input-recycling Convolutional Array of Neurons

Being a 3D-array, the internal architecture and routing of a computation engine could be quite complicated. Additional challenge is how to exploit data reuse, in particular the spatial reuse along the r- and c-loops of Figure 1(a).

Figure 3 illustrates our proposed ICAN architecture, consisting of an *input reuse network* and a *compute tile*. In addition, *shape adapter* is needed to interface with the input buffer. Our compute tile is a 3D array of $T_M T_R T_C$ MAC units with no connection among the elements. The *input reuse network* is a set of registers connected in a 2D-torus[1] interconnect. One register is connected to T_M MAC units of the compute tile, which works in a SIMD fashion. The input reuse network can be loaded very quickly from the input buffer, and provide input data for the connected MAC units during the next K^2 cycles. This can be done by making each value traverse its (K, K) neighbors—for instance, by shifting the register values horizontally first (say, to the west) for the first $(K-1)$ cycles, and then vertically for one cycle, and then reverse-horizontally (to the east) for the $(K-1)$ cycles, and so on. Note that the entire 2D array is shifted simultaneously as in a systolic array, which simplifies control.

To correctly implement the computation of HW-unrolled iterations in Figure 1(a), we need $(K-1)$ extra registers along the eastern edge and the southern edge of the input reuse network, which are called *guard registers*. Therefore the total number of registers are $T_R T_C + K(T_R + T_C)$ if stride is 1, or $((T_R-1)S+1)((T_C-1)S+1) + K((T_R-1)S+1+(T_C-1)S+1)$ for the stride of S. If the stride is greater than 1, each MAC unit is still connected to just one register[2] and the data in the input reuse network are shifted in the exactly same way. The reuse factor will decrease however, but it is due to our parallelization scheme, not our ICAN design.

A *shape adapter* is needed between input buffer and input reuse network to match between the elements of the two 2D arrays. It is simply a 2D array of isolated registers with muxes for handling zero-padding in boundary tiles. The control inputs for the muxes cannot be shared in general, increasing the complexity and area overhead of the *Read Controller*[3]. On the other hand, the shape adapter is needed only once in K^2 cycles, making its latency easily hidden.

Without the input reuse network, a naïve implementation would require K^2 connections for each subarray of T_M MAC units, requring K^2 times more wiring than ours.

The MAC units accumulate all the product terms until the next time-multiplex point (i.e., at every iteration of the *cc*-loop), at which point all the results are written back to the output buffer in a few cycles, followed by loading the next set of output values from the output buffer (our implementation allows 1-cycle simultaneous load/store).

C. Data Tiles and On-chip Buffers

The amount of memory access can have a huge impact on the achievable system throughput. To minimize memory access, our solution generates designs that can make the most use of the available on-chip memory.

[1]The wrap-around is needed either vertically or horizontally, but not both.

[2]In practice, multiplexers are needed if a DNN has layers with different strides because one accelerator must support all convolutional layers of a DNN.

[3]We find that our Read Controller for designs of Section IV-B uses about 9% of the LUT.

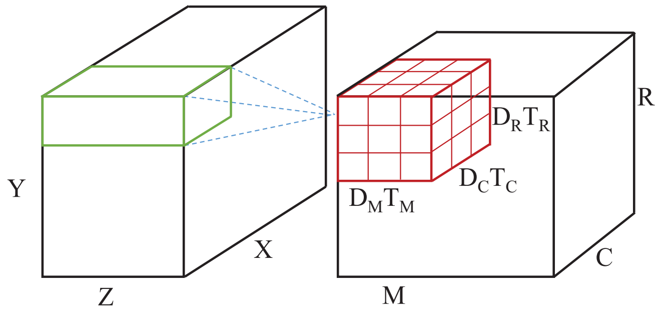


Fig. 4. Data tile (shown in red is the output tile; $D_M = D_R = D_C = 3$).

We introduce *data tile* that is the block of data kept on-chip. We define three data tiles: input, output, and weight tiles. Their sizes are determined by three parameters, D_M, D_R, and D_C as follows (see Figure 4). The *output tile* is simply (D_M, D_R, D_C) times the output of one compute tile, and its size is $B_{out} = D_M T_M \cdot D_R T_R \cdot D_C T_C$ words. The *input tile* is defined as the portion of input data needed to compute the output tile, dropping the Z factor, and its size is $B_{in} = ((D_R T_R - 1)S + K)((D_C T_C - 1)S + K)$. The *weight tile* is defined similarly, dropping both Z and D_M factors, so that its size is $B_w = T_M K K$. The Z factor can be dropped without affecting reuse factor because the z-loop is the outer loop of rr-loop and cc-loop. Similarly the D_M factor can be dropped for weight tile without affecting reuse factor.

On-chip buffers are the physical realization of these data tiles, with double buffering to hide the external memory access latency. The input and weight buffers are single-ported whereas the output buffer is dual-ported. The output buffer has T_M parallel buffers, each of which is $T_R T_C$ words wide, enabling 1-cycle store/load of compute tile output. The input buffer's width is $T_R T_C$ words, and the load latency depends on the amount of data to load, or the value of K and stride. The weight buffer doesn't need high on-chip bandwidth, so we set its width to the memory controller's bus width.

The trip count of each buffer, or how many times they have to be refilled, is given as follows. For output buffer, $\tau_{out} = \left\lceil \frac{M}{D_M T_M} \right\rceil \left\lceil \frac{R}{D_R T_R} \right\rceil \left\lceil \frac{C}{D_C T_C} \right\rceil$; for input buffer, $\tau_{in} = Z \cdot \tau_{out}$; and for weight buffer, $\tau_w = Z \cdot \lceil M/T_M \rceil$. These can be used to calculate the *Computation-To-Communication (CTC)* ratio as in [8].

D. Optimal Parameter Selection

Finding the best combination of compute tile and data tile parameters is an optimization problem. The objective is to maximize the attainable performance, which is the minimum of computation rate (in GOPS) and *bandwidth* · *CTC*, where *bandwidth* is the external memory bandwidth, which depends on the clock speed and the word width.

We have constraints on DSP and on-chip memory usage. On-chip memory (i.e., BRAM) usage constraint is that the sum of the three data tiles' sizes must be within half the available BRAM size (half because of double buffering). The DSP usage

TABLE II
SYNTHESIS RESULT OF ICAN WITH (T_M,T_R,T_C) = (11,7,7)

LUT	FF	DSP	BRAM
9.22%	7.28%	96.25%	52.71%

constraint is that $T_M T_R T_C \leq$ *#MAC units*, where *#MAC units* comes from (1).

We solve the problem using an exhaustive search, which takes less than a minute on a modern workstation using a single-thread execution for each of the designs in Section IV-B.

IV. EXPERIMENTS

We now present (1) synthesis result of our ICAN architecture for 32-bit fixed-point precision, (2) performance comparison with previous work, (3) 32-bit floating-point performance comparison, and (4) DNN size scalability results.

A. Synthesis Result

We have implemented our DNN accelerator in Verilog and synthesized it for Virtex-7 XC7VX485T-2 on a VC707 FPGA board. Vivado 2015.2 is used for simulation and synthesis. The output buffer is implemented with *simple* dual-port BRAMs, which have very small overhead over the single port version. For the memory controller bus we use the AXI Multi-Ported Memory Controller (MPMC) generated by MIG IP block of the tool.

Table II shows the synthesis result for the ICAN architecture implemented using 32-bit fixed-point MAC units each containing 5 DSP slices. We use the optimal architecture parameters supporting all the convolutional layers of the Alexnet [1] under DSP resource constraint set at 2,700 slices. This design can achieve 147.82 GOPS performance throughput. The critical path of the design are the MAC units, for which the operating frequency of 160MHz is achieved, allowing the external memory bandwidth of 6.2 GB/s. The LUT (look-up-table) consumption by ICAN containing 539 MAC units is less than 10%, most of which is due to input reuse network. This synthesis report does not include the shape adapter as most of the shape adapter is implemented in the read controller. For the above design point, we have also synthesized on-chip buffers separately, whose optimal size is found to be (D_M, D_R, D_C) = (18,2,2). The BRAM usage (including double buffering) is less than 60%.

B. Compute Scalability: Exploring Different MAC Options

We evaluate a number of different MAC options to see the scalability of our architecture for higher compute resources. By reducing the precision we can design a MAC using 1 to 5 DSP slices, enabling up to 5x more MAC units. These are all fixed-point DSPs of either 32-bit or 16-bit. For each case we determine the maximum memory bandwidth based on the operating frequency of each MAC unit. For comparison we find the optimal tile sizes for the previous work [8] as well as for ours, considering both compute and memory limit.

Figure 5 shows the performance comparison result. As one can see, the attainable performance gap between ours and the previous work [8] goes as high as 3.41x for 1 DSP-per-MAC case. The graph also shows the computation roof, which is the maximum achievable performance for unlimited memory bandwidth. For our design, the performance lies close to the computation roof.

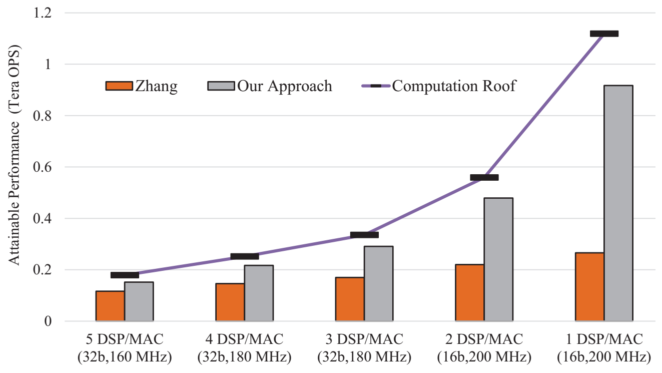


Fig. 5. Attainable performance for different MAC options.

TABLE III
DSP UTILIZATION COMPARISON FOR FIGURE 5

#DSPs/MAC	5	4	3	2	1
DSP utilization of [8]	91.43	91.43	96.00	99.43	82.89*
DSP utilization of Ours	100.0	96.00	96.00	98.00	98.00

*The previous work [8] uses adder trees to do summation along the Z direction. About half of the adders cannot be combined with multipliers *in the one DSP-per-MAC case*; we assume LUTs are used to implement them.

In all the cases except for 1 DSP-per-MAC, both designs are computation-bound. In other words, the difference due to our better on-chip memory management does not play a role except for the 1 DSP-per-MAC case; the performance is determined simply by the product of MAC utilization and DSP utilization (with weighting factors due to differences in the amount of computation among layers). Table III compares the DSP utilization of the two schemes, which also implies that the average MAC utilization (=Performance ratio / DSP utilization) is quite higher for our scheme. In the 1 DSP-per-MAC case, the DSP utilization could be higher if not for the memory bandwidth limit.

Another key factor for this performance difference is that the MAC utilization of the previous work is substantially lower in layer 1 (our architecture has almost the same MAC utilization across all layers). This is because for the first layer, $(Z, M) = (3, 48)$, which means that in their architecture the change in number of MAC units beyond (3×48) doesn't help increase the performance. This may be the peculiarity of the first layer, but most CNNs necessarily have very small numbers of input/output feature maps and very large input/output sizes (i.e., horizontally/vertically) in layer 1. In fact, for later layers (e.g., layers 4 and 5) we do observe that our architecture has lower performance than the previous work in the 5 DSP-per-MAC case.

C. Comparison with the Previous Work

We compare our design approach to that of [8] for floating-point MAC implementation as well, though in practice 16-bit fixed-point implementation (i.e., one DSP-per-MAC) gives enough precision in terms of recognition accuracy for many convolutional neural networks including AlexNet. To evaluate the MAC utilization we keep the DSP utilization same (limited to 80%) as that of [8]. For fare comparison we also use the same operating frequency and memory bandwidth limit as in case of [8]. We find that our MAC utilization is 1.22x (22% higher) compared with that of the previous work, resulting in the performance of 80.78 GFLOPS vs. 66.3 GFLOPS.[4]

D. Size Scalability: Results for Different DNN Sizes

The CNN topology affects the tile size selection. We evaluate our proposed general methodology for different synthetic CNN topology and compare with the previous work for different type of MAC units. We scale down the AlexNet to 10 different CNN topologies. The number of output feature maps, the size of input image, and stride are changed. For each scale-down, we reduce the image size and number of output feature maps by 10% while keeping the input image depth fixed to 3 (i.e., RGB image). In AlexNet, the stride for all layers is 1 except for the first layer, which we do not change. We change the $stride$ of the first layer sequentially as we scale down the net. The sequence of stride is chosen as (4,4,3,3,3,2,2,1,1,1) for 10 different CNN topologies from the largest (*CNN1*) to the smallest size (*CNN10*). We keep the same kernel size, and the input feature maps and row/column of input/output feature maps are determined according to the rule of the convolution operation, stride, and max-pooling architecture of the original DNN.

For each of the 10 CNN topologies we show our performance gain in terms of attainable performance ratio in Figure 6. This comparison shows that our proposed architecture template is scalable across different CNN sizes. This is because we can reach the maximum CTC by utilizing all the available on-chip memory while in the previous work there is always (M/T_M) memory access overhead for input buffer. As a consequence, the attainable throughput of the previous approach is limited by the memory bandwidth, which in turn reduces the DSP utilization—to less than 40% for the two smallest graphs—whereas the DSP utilization for our architecture remains higher than 90% for all sizes.

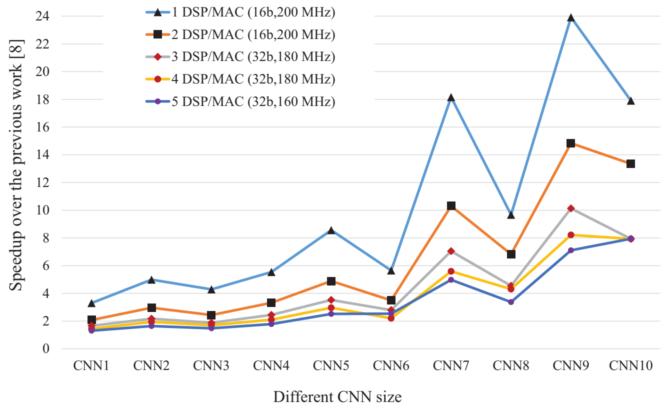


Fig. 6. Performance ratio of ours over the previous work for different CNN sizes.

V. Conclusion

In this paper we presented *ICAN*, our novel accelerator architecture tailored for convolutional neural networks. ICAN is a 3D compute tile that is highly efficient yet flexible for a range of convolutional layer shapes. Being a 3D compute tile, it is a more natural fit for the convolutional layer computation, where the input and output are given in 3D arrays. To address the challenge of complex internal wiring and complex input reuse patterns, we propose an Input Reuse Network that is a simple 2D mesh-like array of registers. Our evaluation mapping the convolutional layers of a real-life DNN demonstrate that our accelerator can not only achieve 22% higher performance on a Virtex-7 FPGA compared with the previous work, but our designs are far more scalable in terms of compute resources and DNN size.

[4]The number 66.3 GFLOPS is based on our estimation, and higher than what is reported in the paper (=61.62) but matches with the number from other sources.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[2] Dan C. Cireşan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jürgen 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*, IJCAI'11, pages 1237–1242. AAAI Press, 2011.

[3] C. Farabet, C. Poulet, J.Y. Han, and Y. LeCun. Cnp: An fpga-based processor for convolutional networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 32–37, Aug 2009.

[4] Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, and Hans Peter Graf. A programmable parallel accelerator for learning and classification. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 273–284, New York, NY, USA, 2010. ACM.

[5] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 247–257, New York, NY, USA, 2010. ACM.

[6] M. Peemen, A.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*, pages 13–19, Oct 2013.

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

[8] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM.