Design Space Exploration of FPGA Accelerators for Convolutional Neural Networks

Atul Rahman*, Sangyun Oh†, Jongeun Lee†‡ and Kiyoung Choi§
*Samsung Electronics, Suwon, South Korea
†School of Electrical and Computer Engineering, UNIST, Ulsan, South Korea
§Dept. of Electrical and Computer Engineering, Seoul National University, Seoul, South Korea

***Abstract*—The increasing use of machine learning algorithms, such as Convolutional Neural Networks (CNNs), makes the hardware accelerator approach very compelling. However the question of how to best design an accelerator for a given CNN has not been answered yet, even on a very fundamental level. This paper addresses that challenge, by providing a novel framework that can universally and accurately evaluate and explore various architectural choices for CNN accelerators on FPGAs. Our exploration framework is more extensive than that of any previous work in terms of the design space, and takes into account various FPGA resources to maximize performance including DSP resources, on-chip memory, and off-chip memory bandwidth. Our experimental results using some of the largest CNN models including one that has 16 convolutional layers demonstrate the efficacy of our framework, as well as the need for such a high-level architecture exploration approach to find the best architecture for a CNN model.**

I. Introduction

Convolutional Neural Networks (CNNs) are used for a broad range of machine learning problems such as object classification and semantic segmentation [1] [2]. Their state-of-the-art recognition performance as well as very high computational complexity makes them a very attractive target for hardware acceleration. In particular, FPGAs, by virtue of reprogrammability, have been a platform of choice for many large-scale CNNs [3]–[8].

The problem of accelerating convolutional neural networks hinges on that of accelerating convolutional layers as they account for most of the computational complexity.

A convolutional layer transforms a number (Z) of input feature maps into a number (M) of output feature maps, where each feature map is a matrix, or a 2D array. One output feature map (B_m) is generated by first performing 2D-convolution between each of the input feature maps (A_z) and a weight matrix ($W_{m,z}$), and then summating the results: $B_m = \sum_z W_{m,z} * A_z$, where $*$ is 2D-convolution. Hence the computation can be described as a 6-deep nested loop as shown in Figure 1.

```
for (m = 0; m < M; m++)
  for (r = 0; r < R; r++)
    for (c = 0; c < C; c++)
      for (z = 0; z < Z; z++)
        for (y = 0; y < K; y++)
          for (x = 0; x < K; x++)
            B[m][r][c] += W[m][z][y][x] × A[z][Sr + y][Sc + x];
```

Fig. 1. Computation of a convolutional layer (adding bias is omitted).

A. Design Considerations

Optimal design of an accelerator involves a number of considerations, among which there are three fundamental aspects. The first is the shape and size of the compute array. The shape of a compute array is the set of dimensions along which the compute array does vectorization. Though a convolution layer includes copious amounts of MAC (multiply-and-accumulate) operations, being distributed in a 6D space, the length along each dimension may not be very large compared with the number of MAC units available on a modern FPGA. This is a problem because, if the number of MAC units does not divide the number of MAC operations along a dimension, some of the MAC units must be idle at some times, lowering resource utilization. For instance, if the Z-loop with Z = 210 is vectorized using an array of 100 MAC units, every third invocation of the MAC array will have just 10% utilization, with the overall utilization of 70% (= 210/300) only. This *internal fragmentation* in the usage of MAC units, is why MAC arrays of different shapes and sizes may lead to very different performance for different layers and CNNs.

The second has to do with data transfer. The size of feature maps is often much larger than that of on-chip memory on FPGAs, which necessitates *loop tiling*, also known as *loop blocking*, to avoid repetitive reloading of same data. Maximizing data reuse, which may be critical to achieving optimal performance for certain layers, requires a careful selection of loop blocking parameters.

The third is execution order. Since one MAC array can handle only so many MAC operations at a time, we must invoke the MAC array many times to cover all the MAC operations in a convolution layer. This again can be represented as a 6D nested loop, and different orders of loops can result in different amounts of data transfer size and performance.

These three aspects are inter-dependent, making it difficult to answer even some basic questions, such as what is the best MAC array shape for a given CNN model, independently of each other. In this paper we present a Design Space Exploration (DSE) framework that can guide system architects in making early decisions in designing optimal CNN accelerators.

B. Contributions

This paper makes the following contributions. First we propose a DSE framework for CNN synthesis that considers a large set of architectural options, including MAC array shapes with all their possible sizes, together with all blocking parameters and all loop orders. This allows us, for the first time, to quantitatively compare different MAC array shapes and sizes for a given CNN layer or model, among other things.

Second, in order to perform DSE using the framework, we need a method to evaluate every point in the design space in a universal way. Instead of developing an optimizing synthesis system of the full CNN accelerator for each design point, which would require enormous

This research was supported by Nano·Material Technology Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (2016M3A7B4909668).
*This work was done while the author was a student at UNIST.
‡J. Lee is the corresponding author of this paper (E-mail: jlee@unist.ac.kr).

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

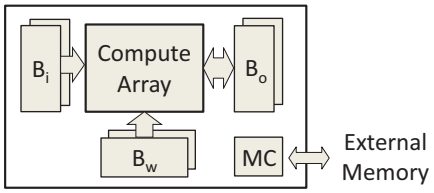

Fig. 2. A CNN accelerator consists of a compute array and three double-buffered on-chip buffers.

engineering effort, we develop a set of methods to characterize and evaluate different architectural options in terms of the three aspects mentioned above, viz., computation, data transfer, and execution order. In particular, for compute time and compute resource usage (i.e., DSP blocks) we provide an accurate model and validate its accuracy through HLS-driven RTL synthesis. For data transfer part, we provide a lower-bound on data transfer time based on bandwidth analysis as it is extremely hard to find out both optimal and implementable designs, which are continuously improved with human ingenuity. For execution order we use simulation, which is accurate.

Third, while we assume, like the previous work, that the FPGA is not dynamically reconfigured across layers of a CNN model, we do allow execution order to be different across layers. This helps simplify our DSE evaluation engine in addition to extending the design space and consequently helps achieve slightly higher performance.

Fourth, we present our DSE results involving 5 different large-scale CNN models and multiple FPGA architectures.

There are some limitations. First, we do not vectorize K-loops (see Figure 1), since K is relatively small and often varies across layers; however, fully unrolling them as in [4], [7] is a trivial extension. Second, we assume that the clock speed is given by the user, which can be justified since we use DSP blocks (e.g., DSP48E in Xilinx FPGAs) to implement MAC operations, and in that case the critical path lies in the MAC units, and is not very sensitive to the size or shape of the array. Third, a recent proposal [4] uses LUTs (look-up tables) to synthesize some MAC units. While it can help create larger MAC arrays and thus achieve higher performance, the drawback is that estimating the number of MAC units available becomes very difficult, jeopardizing parameter optimization. Indeed our experiments in Section IV-F demonstrate that our framework, though not using LUTs, can still generate solutions competitive with that of [4] that uses LUTs thanks to our exhaustive DSE.

Through our exploration we find that the variation in performance is indeed great among different architectures in a quite unpredictable way. We also confirm that the best MAC array shape is not necessarily fixed, but depends a lot on the CNN model and to a lesser degree on the target FPGA. Finally even with the same MAC array shape our framework can often find (potentially) better architectures than the previous work, again thanks to its extensive exploration capability.

The rest of the paper is organized as follows. In Section II we present our DSE framework. In Section III we present our universal evaluation method for DSE. In Section IV we present our experimental results, discuss related work in Section V, and conclude in Section VI.

II. Our Exploration Framework

A. Accelerator Overview

Our accelerator model consists of a compute array and three on-chip buffers as illustrated in Figure 2. A *compute array* is an array of compute elements, which are more or less homogeneous blocks containing multipliers and adders. In this paper we use the terms *compute array* and *MAC array* interchangeably. A compute array implements the loop body computation of the kernel expanded along some dimensions of the iteration space. The set of those dimensions is called the *shape* of a compute array, which dictates the internal structure of the compute array.

Buffers are scratchpad memories that partially or fully store the input/output arrays and weight parameters. If partially stored, it is backed up by the external memory (such as DRAM), and data must be transferred between buffers and the external memory, which happens simultaneously with computation thanks to double buffering. Thus a buffer must be double-buffered unless it need not be backed up by the external memory.

We assume that multipliers and adders needed to execute the loop body operations of the kernel are implemented using DSP blocks. Likewise buffers are assumed to be implemented using RAM blocks. We assume that the maximum sustained memory bandwidth is given, which can be obtained empirically by simulating the memory controller(s) for long enough time.

Completing one convolutional layer requires performing the many multiply-and-accumulate (MAC) operations of the kernel, which takes many cycles even if all the DSP blocks of an FPGA are used. Thus layers of a CNN are done one after another, and we use the same FPGA configuration for all layers (in other words, no runtime reconfiguration).

```
1  for (m2 = 0; m2 < M; m2+=B_M)
2  for (r2 = 0; r2 < R; r2+=B_R)
3  for (c2 = 0; c2 < C; c2+=B_C)
4  for (z2 = 0; z2 < Z; z2+=B_Z)

5  for (m1 = m2; m1 < min(M, m2 + B_M); m1+=T_M)
6  for (r1 = r2; r1 < min(R, r2 + B_R); r1+=T_R)
7  for (c1 = c2; c1 < min(C, c2 + B_C); c1+=T_C)
8  for (z1 = z2; z1 < min(Z, z2 + B_Z); z1+=T_Z)
9  for (y = 0; y < K; y++)
10 for (x = 0; x < K; x++)

11 for (m = m1; m < min(M, m1 + T_M); m++)  // unroll
12 for (r = r1; r < min(R, r1 + T_R); r++)  // unroll
13 for (c = c1; c < min(C, c1 + T_C); c++)  // unroll
14 for (z = z1; z < min(Z, z1 + T_Z); z++)  // unroll
15     B[m][r][c] += W[m][z][y][x] × A[z][Sr + y][Sc + x];
```

Fig. 3. Convolutional layer computation, after loop blocking.

B. Design Space

Building on the previous work's [3] intuition that different accelerator architectures can be generated from differently transformed loop nests, we explore *all* loop nests created by loop unrolling, loop interchange, and loop blocking (also known as loop tiling). Many of the transformations are duplicate from the performance perspective, and there are only a finite number of cases we need to deal with.

The most general version, before loop interchange, is obtained by tiling each loop level twice and fully unrolling the inner one. One such loop nest is shown in Figure 3, where K-loops are not tiled since K is typically small and varies across layers (When K is constant, unrolling them is trivial). In the figure the four innermost loops (lines 11–14) are fully unrolled and implemented as the compute array. Thus they can be regarded as an atomic operation, which we call *compute-array operation*. Since one compute-array

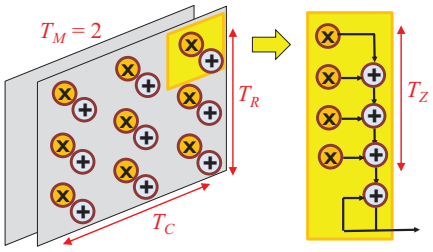

Fig. 4. A 4D compute array with the configuration of (T_M, T_R, T_C, T_Z) = (2, 3, 3, 4).

operation is equivalent to $T_M T_R T_C T_Z$ MACs, and it needs to be invoked roughly $K^2\lceil M/T_M\rceil\lceil R/T_R\rceil\lceil C/T_C\rceil\lceil Z/T_Z\rceil$ times to complete one convolution layer.

The difference between the four outermost loops and the six middle loops is that the latter performs its computation using on-chip buffers only, as is typically the case with conventional loop blocking. In other words we would like to set the *B-parameters* (i.e., B_M, B_R, B_C, B_Z) such that the required buffers can fit in the available on-chip memories of an FPGA. We require that a B-parameter be a multiple of a corresponding T-parameter due to the granularity of compute-array operations. Then the number of iterations in the outermost loops determines how many times each buffer must be reloaded, also called *trip count*, with one caveat—some buffers need not be reloaded in certain iterations due to data reuse. For instance, an increment of z_2 at line 4 does not change the subset of the output array that needs to be on-chip, making it unnecessary to reload the output buffer along the z_2-loop except in the first iteration.

Clearly loop interchange makes a difference only among the four outermost loops (lines 1–4). For the K-loops (lines 9–10) we allow full blocking, i.e., $B_K = K$, since K is relatively small.

This leads to the definition of a configuration as an 8-tuple including all the *T-parameters* (also called *unroll parameters*) and *B-parameters* (*blocking parameters*), plus the loop order of the four outermost loops. Since the loop order is software-controlled, it need not be the same across layers. Exploration of *nonuniform* as well as uniform loop orders, which has not been done before, is another minor feature of our DSE. The design space we consider in our DSE framework does not cover every accelerator design conceivable, but is a superset of those of the previous work all combined.

III. Evaluation of a Design Point

The objective of our exploration is to minimize cycle count, and when the cycle count is equal, minimize the bandwidth requirement. As for the constraints we consider the number of DSP blocks, the amount of on-chip RAM, and the off-chip memory bandwidth. Thus it is necessary to evaluate any given configuration in terms of cycle count, bandwidth requirement, and resource usage (DSP and RAM). In the following we use superscript to denote the layer parameter of a particular layer. For example S^l is the stride value of layer l.

A. Computation: Compute Time and DSP Usage

1) DSP Usage: We show that the optimal number of DSP units to implement a given shape and size of a compute array, as specified by a 4-tuple (T_M, T_R, T_C, T_Z), is indeed $T_M T_R T_C T_Z$, and that it achieves the maximal throughput of $T_M T_R T_C T_Z$ MAC ops/cycle. It is obvious that we need at least $T_M T_R T_C T_Z$ DSP units. The question is whether it is always possible to implement a compute unit with the maximal throughput using only the minimum number of DSP units.

Our solution is illustrated in Figure 4, which can implement any of the compute array shapes. If $T_Z = 1$, the MAC pipeline in the figure is reduced to a pair of multiplier and adder, which is then replicated $T_M T_R T_C$ times. In this case it is obvious that we need $T_M T_R T_C$ DSP units only, since one DSP unit can implement a multiplication and an addition with the initiation interval of 1 cycle.

If $T_Z \geq 2$, we need the pipeline structure. In this case it is not obvious whether we need an extra DSP unit for the last accumulator. But our experiment in Section IV-B shows that no extra DSP unit is needed, and that the accumulator can be paired with the first multiplier, while achieving the minimum initiation interval of 1 cycle.

The pipeline structure in Figure 4 can support bias addition and initial value without extra DSP units, by modifying the feedback path in the accumulator to receive either the accumulator value, bias, or initial value. Initial value is needed if the output feature map value must be updated, e.g., when $Z > T_Z$.

2) Compute Time: The compute array has the initiation interval of 1 cycle, and therefore *compute time*, or the total number of cycles to complete all the MAC operations of a convolution layer l is:

$$T^l_{comp} = (K^l K^l D_M D_R D_C D_Z + T_Z - 1) \lceil M^l/B_M\rceil\lceil R^l/B_R\rceil\lceil C^l/B_C\rceil\lceil Z^l/B_Z\rceil,$$

where $D_M = B_M/T_M$ (others are similarly defined) and $(T_Z - 1)$ is due to pipeline filling. The ceiling operator here is what causes internal fragmentation, and also why simply doubling the size of a MAC array may not double the computation rate, let alone the overall performance.

B. Data Transfer

1) Effective Buffer Size and SRAM Usage: There is often a tradeoff between physical buffer size and bandwidth. Bandwidth requirement can be reduced if we can afford larger on-chip buffers. Thus while most previous work except [6] did not consider maximizing the on-chip memory usage, optimization of buffer size could be critical to achieving the highest performance.

One tricky aspect of physical buffer size, unlike bandwidth calculation (see the next section), is that we must know the width and depth of buffers (as opposed to the capacity), and that we must synthesize buffers with the dimensions that can support all layers (i.e., we must use the maximum width and depth across all layers).

The width can be determined from the requirement that the compute array must be able to read (and write) a certain number of words simultaneously in order to sustain the initiation interval of 1 cycle. For example, the output buffer must provide the bandwidth of $T_M T_R T_C$ words/cycle. For the input buffer, due to the inherent data reuse in the convolution operation, there is a large gap between naïve vs. optimized versions. We use the lowest value assuming full data reuse, which can be implemented using a structure like the one proposed in [5].

$$B_o^{wdth} = T_M T_R T_C$$
$$B_w^{wdth} = T_M T_Z$$
$$B_i^{wdth} = \max_l t^l_Z (S^l(t^l_R - 1) + K^l)(S^l(t^l_C - 1) + K^l)$$

where the unit is words. For B_i^{wdth} we use *effective unroll parameters*, t^l_Z, t^l_R, t^l_C, which are defined as $t^l_Z = \min(T_Z, Z^l)$ and analogously for the others, instead of their corresponding unroll parameters so as to lower the physical buffer size requirement. For instance, an optimized design for AlexNet [1] has T_Z value (=7) that is greater than Z^l in the first layer which is only 3. Coincidentally S^l and K^l in the first layer are also very high, leading to huge savings

in the required buffer size as compared to using the original unroll parameters, and consequently increased performance.

The required capacity of a buffer for each layer (the numerator part in the following equations) can be similarly determined, from which one can find buffer depth for each layer as follows.

$$B_o^{dpth} = \max_l \lceil b_M^l b_R^l b_C^l / B_o^{wdth} \rceil$$
$$B_w^{dpth} = \max_l \lceil b_M^l b_Z^l K^l K^l / B_w^{wdth} \rceil$$
$$B_i^{dpth} = \max_l \lceil b_Z^l (S^l (b_R^l - 1) + K^l)(S^l (b_C^l - 1) + K^l) / B_i^{wdth} \rceil$$

The effective blocking parameters, b_M^l, b_R^l, etc., are defined similarly, e.g., $b_M^l = \min(B_M, M^l)$. Since blocking parameters are usually very large (certainly much larger than unroll parameters), using the effective versions is critical to obtaining realistic buffer sizes.

Finally physical buffer size requirement is simply the product of width and depth of the buffer. The sum of physical buffer sizes of all three buffers must be no greater than the available on-chip memory, which is the SRAM constraint.

2) Data Transfer Size and Bandwidth: To calculate bandwidth requirement, we need to know the trip count, or the number of buffer reloadings (see Section III-C). Then the amount of external data transfer needed for each buffer, for each layer, is simply the required capacity of the buffer (defined above) multiplied by the trip count. For example, for output buffer it is $\tau_o^l b_M^l b_R^l b_C^l$, where τ_o^l is the trip count of output buffer for layer l.

Then the total data transfer for each layer is the sum of the data transfers of all three buffers, from which one can calculate T_{dt}^l, the data transfer time for layer l (in cycles).

$$T_{dt}^l = \mathit{DataTransSize}^l \cdot f_{CLK} / \mathit{BW\_limit} \quad (1)$$

Finally the execution time ($T = \sum_l T^l$) and the required bandwidth of the design are given as:

$$T^l = \max(T_{comp}^l, T_{dt}^l) \quad (2)$$
$$\mathit{BW} = \max_l \mathit{DataTransSize}^l / T^l. \quad (3)$$

C. Execution: Calculating Trip Counts

Previous work uses predefined formulas to calculate trip counts, which is fine only if the compute array shape and the loop order are all fixed. In our case, coming up with formulas for all of the 360 combinations ($= 15 \times 4!$) is tedious and error-prone. In order to find accurate trip counts for any design point, we use simulation. Our simulation finds the number of reloadings necessary in the four outermost loops (lines 1–4 of Figure 3) for any given layer, buffer, and loop order. It takes as input all the four layer parameters (M, R, C, Z) and four blocking parameters.

Like the previous work (e.g., [3]), the trip count for the output buffer is multiplied by 2, since it is both read and written, unless it is loaded only once. One effect that is overlooked in the previous work including [3] is that some of the loops in lines 1–4 of Figure 3 may disappear due to having one iteration only. Our simulation can find out exact trip counts, by keeping track of the range of block addresses needed in each iteration and incrementing the trip counts only if the block addresses change between consecutive iterations of the four outermost loops.

While our evaluation covers most important aspects, viz., computation, data transfer, and execution, it does not guarantee exactness or feasibility, which is due to the data transfer part. First our evaluation does not consider data alignment, which uses LUTs as well as many registers, and could make some design points infeasible. Second our buffer size estimation is idealized, giving the minimum size requirement; the actual synthesized buffer sizes tend to be larger. On the other hand, our evaluation gives the upper bound of the performance for any given design point, which can be useful for a limit study and in guiding designers making early decisions.

TABLE I
CNN MODELS

CNN	Description	#Conv. Layers	#MAC Ops
CNN1	AlexNet [1]	5	1.33 B
CNN2	Speed sign recognition [6]	3	5.40 B
CNN3	Street scene parsing [9]	4	13.10 B
CNN4	ConvNet-A (VGG11) [2]	8	14.97 B
CNN5	ConvNet-E (VGG19) [2]	16	39.01 B

TABLE II
CHANGE IN RESOURCE USAGE

Last tap	#Cycles	T_{comp} (pred.)	#DSP	#LUT	f_{CLK}
Accumulator	55,765	55,360	2,688	56,418	205
AND gate	55,765	55,360	2,688	63,586	205

IV. EXPERIMENTS

A. Experimental Setup

We use five CNN models listed in Table I, which are among the largest reported in the literature. Each CNN contains 5∼16 convolution layers, with 36 layers used in total.

The data type of a MAC operation affects the frequency and effective bandwidth (BW) limit, which we obtain from our RTL synthesis and simulation results using Vivado 2016.1 targeting Xilinx Virtex7-485T FPGA on a VC707 evaluation board. We use 16-bit fixed-point precision as 16-bit results are shown to be accurate enough [4]. From our RTL synthesis results of our ZM- and MRC-shaped MAC array designs, with parameters optimized for the CNN1 case, we have verified that 16-bit MAC arrays can achieve 200 MHz clock speed, which is the maximum speed of our DDR memory controller. The Virtex7-485T FPGA contains 2,800 DSP slices and over 4.7 MB on-chip memory (Block RAM). For 200 MHz the off-chip bandwidth from RTL simulation is found to be at least 9 GB/s (hence our BW limit), which is the same BW per clock frequency as in previous work [3], [5].

We perform DSE in an exhaustive way, employing some obvious heuristics for pruning. In reporting results we often use performance numbers in GOPS (10^9 Ops/s) or Tera OPS, which are converted from cycle count numbers for easier comparison among different CNN models. One MAC is counted as 2 operations.

B. Validation of Our Compute Array Characterization

We have validated our DSP count and cycle count model for compute array in three ways. First, we generate two designs for $(T_M, T_R, T_C, T_Z) = (14, 8, 8, 3)$; one as in Figure 4 and the other with the accumulator replaced with an AND gate. This is to see the effect of the accumulator on the DSP usage. We use the parameters for the first layer of VGG19, and $(B_M, B_R, B_C, B_Z) = (42, 64, 64, 3)$. Our result summarized in Table II shows that the DSP count is not changed at all (and it matches the predicted value) and the AND-gate version uses significantly more LUTs (Look-Up Tables), implying that our architecture in Figure 4 uses the minimal number of DSP units. Additionally it demonstrates the accuracy of our cycle count and DSP count estimation.

Second, we have generated 15 different configurations, synthesized the compute arrays using Vivado HLS, and compared the DSP usage

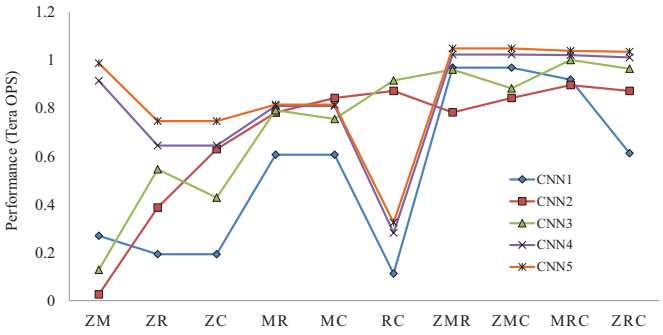

Fig. 5. Performance vs. MAC array shape (100% resources).

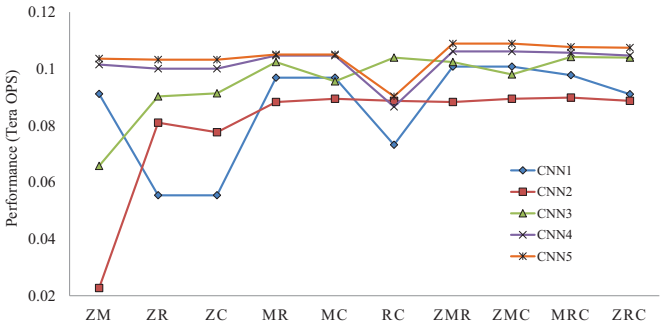

Fig. 6. Performance vs. MAC array shape (10% resources).

with our DSP count model. We find that in all the cases the DSP count exactly matches the prediction by our model, while meeting the initiation interval of 1 cycle and the target clock frequency of 200 MHz. Third, for 4 out of the 15 configurations that use both R and C dimensions, we have completed the accelerator design including computation and data transfer, and measured the cycle count after RTL synthesis by Vivado HLS. In all the 4 cases the cycle count matches our prediction model, with less than 1% error.

C. Effect of CNN Model

A common misconception in the evaluation of CNN accelerators is that an architecture that works well for one CNN will work equally well for other CNNs. This fallacy is often manifested in comparisons between accelerators using GOPS ratings obtained from different CNNs, which seems quite prevalent perhaps because of the convenience it offers. After all, one may think, all CNN models belong to the same application—the convolutional neural network—with the same computation and communication patterns, so why not?

To show that it is a fallacy, we perform an exploration for the five CNN models with respect to different MAC array shapes, which are shown on the x-axis of Figure 5. MAC array shape ZM, for instance, means that $T_R = T_C = 1$ and only T_Z and T_M are explored. The B-parameters and the loop order are always explored. The best performance number for each MAC array shape is plotted.[1]

First we note that the performance of architectures depends heavily on the CNN model. The RC shape, for instance, is the best among 2D shapes when running CNN3, but is indeed the worst for CNN1. (Note that what we refer to as *architecture* is not a specific accelerator design but an architecture template, whose parameter values are optimized through exploration.) There is no consistent winner among architectures, with ZM being the most polarizing: it tends to be either the best or the worst 2D architecture. Even among 3D architectures, there is no one that is optimal for all CNN models. For instance, MRC is overall very good but is not optimal for CNN1.

The graph also shows that the 3D shapes are generally better than 2D shapes, which is partially because a 3D shape can always degenerate to a 2D shape if 3D does not give a competitive edge (i.e., ZMR becoming MR when $T_Z = 1$). But a close examination reveals that there is indeed a significant gap between 2D and 3D performance, meaning that the 3D shapes do have a distinct advantage over 2D shapes.

[1]In our evaluation we assume that the clock frequency is the same across MAC array shapes, which is based in part on the results of Section IV-B. But some simpler MAC array shapes could achieve higher clock frequency than our target frequency. Such constraints, once quantified, can be incorporated into our exploration framework.

TABLE III
PERFORMANCE (IN GOPS) IMPROVES AS EXPLORATION EXPANDS

CNN	[3]	+LO	+FB	+MAS	Optimal
CNN1	260.79	267.31	269.39	967.65	MRZ
CNN2	25.58	25.58	25.58	895.53	MRC
CNN3	127.89	127.89	127.89	1,000.45	MRC
CNN4	887.78	914.48	914.48	1,023.32	MRCZ
CNN5	963.25	987.26	987.26	1,048.72	MRZ

Note: LO (Loop Order), FB (Full Blocking), MAS (MAC Array Shapes).

D. Effect of Hardware Resources

The previous graph shows that the best architecture varies depending on the CNN model. But is this *best architecture* the same regardless of hardware setting or will it change? To see this, we make a small change in the hardware setting, which is to reduce the amount of FPGA resources to 10%, and do the same exploration again. This is to simulate a case where 10 similar accelerators share the same FPGA chip. Thus we reduce the number of DSPs, the size of on-chip memory, and the off-chip bandwidth by 10 times, but not the clock speed of the FPGA.

The results are shown in Figure 6. Since every resource type has been changed by the same ratio, one may expect nearly an identical trend as in the previous graph. However we see a few important changes. First the gap between 2D shapes and 3D shapes is much less, suggesting that 2D architectures may be a good fit for smaller FPGAs. Second, increasing resources 10 times does not necessarily increase performance 10 times—often it is less. Third, performance of some architectures can vary drastically depending on the amount of resources. For instance, RC, which is one of the lowest performers in the 100% resource case shows good performance at 10% case. While others do not easily change due to hardware changes, our hardware changes are very small. A more extensive study would be needed to confirm how close the tie is between CNN models and optimal architectures.

E. Impact of Different Architectural Options

Our framework has three main architectural options, which jointly determine performance. To see their relative importance on performance we do incremental exploration, expanding architectural options one by one.

Table III summarizes the results. The second column shows the performance of the ZM architecture with limited 2D exploration for blocking parameters and a fixed loop order, which is identical to [3]. From there, we first add loop order exploration, which gives very small performance improvement of about 3% at most. Next we add full blocking parameter exploration, which gives essentially no

TABLE IV
COMPARISON WITH THE PREVIOUS WORK

Features	[3]	[4]	[5]	[6]	This paper
Exploring MAC array shapes*	No (ZM)	No (ZM)	No (MRC)	No (MR, MC)	Yes
Exploring blocking parameters	Limited (2D)	Limited (2D)	Limited (3D)	Full	Full
Exploring loop orders	No	No	No	Yes	Yes

*Note: The K-loops are not considered for vectorization. Even though some [4], [7] have unrolled K-loops, it is full, unconditional unrolling and differs from vectorization (i.e., there is no such parameter as T_K).

TABLE V
COMPARING VGG16 RESULTS ON KINTEX

	Total latency	GOPS	Speed-up
Theoretical [4]	123.10 ms	249.31	0%
Measurement [4]	163.42 ms	187.80	—
Our exploration result	115.15 ms	266.53	+6.9%

improvement. Finally we add MAC array shape exploration, which brings up to 35x performance improvement in one case. The MAC array shape exploration is a unique feature of our DSE framework whereas [6] performs loop order and full blocking parameter exploration only. Our results clearly show that the MAC array shape exploration is a key ingredient to achieving the highest performance on an FPGA consistently for all CNN models, which is around 900∼1,000 GOPS. The last column shows the MAC array shape that is used to achieve the optimal performance. The optimal architecture varies but M and R do appear consistently for all the CNN models.

F. Comparison with the Case That Uses LUTs

One limitation of our exploration framework is that it does not use LUTs for implementing MAC ops. This may result in poorer performance when compared with those architectures [4] that do use LUTs in addition to DSP blocks for MAC ops. In this section we provide a quick comparison, where we use the same CNN model (ConvNet-D [2] with 13 convolutional layers) and the same hardware settings including the number of DSPs (900), off-chip bandwidth (4.2 GB/s), and clock speed (150 MHz) as in the previous work.

Our exploration result (Table V) shows that our best architecture relying on DSP units for most MAC operations is able to achieve 266.5 GOPS, which is faster by about 6.9% even compared with the their *theoretical speed*, 249.3 GOPS. The reason for this surprising result is in part due to (i) the low DSP utilization of their architecture, which is only 89.2%, (ii) not considering the internal fragmentation issue when optimizing architectural parameters, and (iii) our framework's extensive optimization including the MAC array shape. Also possible is that when using LUTs, the difficulty of accurately estimating the usable MAC count may make parameter optimization more difficult.

V. RELATED WORK

Table IV compares some of the most closely related work of this paper. The authors of [3] present a CNN architecture for FPGAs and a parameter optimization methodology based on the well-known roofline model, which is one method to help balance computation time and data transfer time. We also employ the same idea though not using the roofline model itself.

The authors of [4] present an end-to-end design including not just convolutional layers but also fully-connected layers, in addition to a quantization method. They also utilize LUTs to generate some more MACs, which may help achieve higher performance.

While most of the accelerator designs are based on a 2D array of MACs, [5] proposes a 3D array (MRC), which has a high operating frequency due to their input-reuse network. We assume the same architecture and design for the MRC case in our exploration.

One paper [6] discusses blocking parameters and loop orders, and the exploration thereof in a very general nested loop context. A CNN is also used as an example, but their exploration is focused on minimizing data transfer overhead, and not so much on maximizing the performance of a CNN accelerator.

Our work starts from a very different perspective. We build on the previous architectures proposed so far, but our concern is how to help designers find optimal CNN architectures for FPGAs, and how to assure that a design is sufficiently optimal considering key architectural options and important hardware resources available. The key ingredient in optimizing for performance is to explore MAC array shapes as demonstrated in our experiments, but it necessarily requires *simultaneous exploration of all three architectural options*, which adds greatly to the challenge of the problem—not just from the computational complexity perspective but also from that of how to design an exploration framework that can handle all different architectural configurations in a uniform and algorithmic way.

VI. CONCLUSION

We presented an architecture exploration approach targeting CNN accelerators on FPGAs. Unlike previous work which is often tied to a specific architecture, our approach allows for a quick evaluation of many valid architectures which we find essential in finding the best architecture that suits the target application. Our exploration framework is not only extensive but also highly optimizing in terms of utilizing hardware resources such as DSP resources, on-chip memory, and off-chip memory bandwidth. We have demonstrated the efficacy of our framework through experiments using some of the largest CNN models, which also point out the need for a high-level architecture exploration approach such as ours to find the best architectures for CNN models.

REFERENCES

[1] A. Krizhevsky *et al.*, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, pp. 1097–1105.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[3] C. Zhang *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *FPGA '15*, 2015, pp. 161–170.

[4] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *FPGA '16*. ACM, 2016, pp. 26–35.

[5] A. Rahman *et al.*, "Efficient FPGA acceleration of convolutional neural networks using logical-3D compute array," in *DATE '16*, 2016, pp. 1393–1398.

[6] M. Peemen *et al.*, "Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators," in *DATE '15*, 2015, pp. 169–174.

[7] M. Sankaradas *et al.*, "A massively parallel coprocessor for convolutional neural networks," in *ASAP '09*, July 2009, pp. 53–60.

[8] D. Nguyen *et al.*, "Double MAC: Doubling the performance of convolutional neural networks on modern FPGAs," in *DATE '17*, 2017.

[9] C. Farabet *et al.*, "Scene parsing with multiscale feature learning, purity trees, and optimal covers," *arXiv preprint arXiv:1202.2160*, 2012.