Escher: A CNN Accelerator with Flexible Buffering to Minimize Off-Chip Transfer

Yongming Shen
Stony Brook University
yoshen@cs.stonybrook.edu

Michael Ferdman
Stony Brook University
mferdman@cs.stonybrook.edu

Peter Milder
Stony Brook University
peter.milder@stonybrook.edu

***Abstract*—Convolutional neural networks (CNNs) are used to solve many challenging machine learning problems. Interest in CNNs has led to the design of CNN accelerators to improve CNN evaluation throughput and efficiency. Importantly, the bandwidth demand from weight data transfer for modern large CNNs causes CNN accelerators to be severely bandwidth bottlenecked, prompting the need for processing images in batches to increase weight reuse. However, existing CNN accelerator designs limit the choice of batch sizes and lack support for batch processing of convolutional layers.**

We observe that, for a given storage budget, choosing the best batch size requires balancing the input and weight transfer. We propose Escher, a CNN accelerator with a flexible data buffering scheme that ensures a balance between the input and weight transfer bandwidth, significantly reducing overall bandwidth requirements. For example, compared to the state-of-the-art CNN accelerator designs targeting a Virtex-7 690T FPGA, Escher reduces the accelerator peak bandwidth requirements by $2.4\times$ across both fully-connected and convolutional layers on fixed-point AlexNet, and reduces convolutional layer bandwidth by up to $10.5\times$ on fixed-point GoogleNet.

I. Introduction

Convolutional neural networks (CNNs) are being used to solve a wide array of challenging machine learning problems, such as recommendation systems [1], natural language processing [2], and computer vision [3]–[5]. In particular, CNNs achieve unprecedented image object recognition accuracy, setting new records for object detection and classification competitions every year. However, the high accuracy of CNNs comes with a high computational cost. CPUs cannot provide sufficient performance, leading to increasing popularity in the use of hardware accelerators [6]–[9]. Among the accelerators, FPGAs are emerging as an appealing option, as they hold the promise of simultaneous high performance and energy efficiency.

CNNs comprise a series of computation *layers*, where each layer takes the output of the preceding layer as its input. Each layer scales inputs by a set of weights (parameters that are found through learning). A modern CNN contains hundreds of megabytes of weights, leading to a severe off-chip bandwidth bottleneck [7], [10]–[14].

Importantly, although each weight is needed at least once per image, the same weights are used for evaluating every image. State-of-the-art CNN accelerators leverage this by batching the processing of multiple images, bringing weights on chip in blocks and reusing them across batches of inputs. For the fully-connected (FC) layers, where the off-chip weight transfer is a clear bottleneck, batching of images effectively reduces memory bandwidth requirements by a factor of the batch size.

Previous batch processing schemes limit the batch size by connecting it to other aspects of the accelerator's architecture. For example, to support batching, [7] must buffer many outputs on chip, limiting the accelerator to small batch sizes due to constrained on-chip memory capacity. The weight-major method in [11] requires the batch size to match the number of vector dot product units in the accelerator, also limiting the design to small batch sizes. On the other hand, the input-major method in [11] supports large batch sizes, but limits the amount of output data stored per image; this alleviates the weight bandwidth bottleneck, but creates a new bandwidth bottleneck from re-transferring input data multiple times.

We observe that, for a given on-chip storage budget, increasing the batch size necessarily reduces the amount of data buffered per image because data for all images in the batch must be stored. Therefore, increasing the batch size reduces the weight bandwidth, but increases the bandwidth required for transferring input/output data because less on-chip storage is allocated to them. The result is that one must strike a careful balance between weight bandwidth and input/output bandwidth. Therefore, we propose a CNN accelerator design that allows flexible batching by allowing independent adjustment of the batch size and the amount of data buffered per image. This allows a design to operate close to the optimal trade-off point between weight transfer and input/output data transfer, significantly reducing the bandwidth required relative to prior designs.

Moreover, we find that reducing the bandwidth requirement of fully-connected layers exposes a new bandwidth bottleneck: the convolutional layers, for which the prior work has no solution. Unlike previous approaches, our method is flexible enough to also support batching on the convolutional layers. Using the same amount of on-chip storage, our approach enables larger batch sizes without unnecessary penalties in input/output re-transfer. We achieve a significant reduction in the bandwidth requirements of the entire CNN, which includes both the fully-connected and convolutional layers.

In this work, we propose Escher, a CNN accelerator architecture that emphasizes flexibility in controlling how data are buffered on chip. The batch size and the amount of data stored per image can be adjusted independently and can change from layer to layer. For Escher, we develop an optimization strategy that determines the appropriate buffer sizes, per-layer batch size, and per-layer data tiling parameters for all fully-connected and convolutional layers of a CNN. The result of the optimization produces an Escher design that makes near-optimal use of the on-chip buffers to minimize overall accelerator bandwidth requirements.

We demonstrate the effectiveness of Escher by designing FPGA accelerators for three modern CNNs (AlexNet [3], VGGNet-E [4], and GoogleNet [5]) in floating point and fixed point configurations.

On a Xilinx Virtex-7 690T FPGA, using our approach for batching both the fully-connected and convolutional layers, Escher reduces the peak bandwidth requirements by up to $2.4\times$ (AlexNet, fixed point) compared to the state of the art, while using essentially equivalent FPGA resources. Escher also demonstrates that batch processing can reduce the bandwidth requirements of convolutional layers by up to $10.5\times$ (GoogleNet, fixed point). In contrast to conventional wisdom, our results demonstrate that batch processing of convolutional layers is needed to avoid bandwidth bottlenecks. For example, batching both fully-connected and convolution layers can reduce the peak bandwidth consumption by $1.7\times$ (VGGNet-E, fixed point) compared to batching only the fully-connected layers.

II. Background and Motivation

A CNN is a pipeline that takes an input and passes it through multiple stages, called *layers*, producing a vector as the final output. In the context of object detection, the input is an image and the output vector identifies an object found in the image.

Convolutional layers, responsible for feature extraction, compute Y output feature maps from X input feature maps, where each feature map is an $R\times C$ matrix. An output feature map is computed by 3D-convolving the $X \times R_{in} \times C_{in}$ inputs with $X \times K \times K$ filters, stepping with stride S. Critically, there are Y filters, one for each output feature map. Every convolutional layer may have different dimensions, each defined by the tuple *(X,Y,R,C,K,S)*.

Fully-connected (FC) layers, responsible for classification, compute an output feature vector of size Y from an input feature vector of size X. Each value in the output feature vector is the dot product of the input vector and a weight vector. As for convolutional layers, different weight vectors are used to compute each of the values in the output vector. Every fully-connected layer may have different *(X,Y)* dimensions. Critically for our work, a fully-connected layer can also be thought of as a convolutional layer defined by the tuple *(X,Y,R=1,C=1,K=1,S=1)*.

A. The Weight Transfer Bottleneck

Modern CNNs use hundreds of megabytes of filter weights, far exceeding the on-chip memory capacity of any FPGA. To process an image, each weight must be read from off-chip memory at least once. If images are processed in isolation, weight transfer alone will cause severe memory bandwidth bottlenecks.

For example, using a 512-bit 100MHz interconnect between an FPGA accelerator and the memory controller can provide at most 6GB/s bandwidth. For AlexNet, which uses 233MB of weights (assuming 32 bits per weight), this limits the throughput to 26.4 images/s. For VGGNet-E, which uses 548MB of weights, the limit is 11.2 images/s. In practice, peak accelerator throughput is even lower, because some bandwidth is used to transfer input and output feature maps, and because actual DRAM access patterns and refresh limit the peak achievable bandwidth.

B. Batch Processing CNN Layers

To reduce the off-chip bandwidth requirements of weight transfer, images can be processed in batches [7], [11][1]. Buffering data for a batch of images on chip enables transferred weights to be reused for processing all images in the batch. Thus the number of weights transferred is reduced linearly with the batch size.

However, existing designs fail to realize the full potential of batch processing. The Caffeine [11] weight-major method constrains the batch size to the number of vector dot product units in the accelerator. For the CNNs we evaluate, the best throughput is achieved at approximately 70 dot product units, but larger batch sizes are desired to mitigate the weight transfer bandwidth bottleneck. The Caffeine input-major method stores a small amount of output data per image (corresponding to the number of vector dot product units) to enable large batch size. However, a typical fully-connected layer output vector size of 4096 causes each input to be re-transferred approximately $4096/70 = 59$ times, which creates a bandwidth bottleneck due to input feature vector transfers. In [7], an alternative strategy of storing complete output vectors is used, however this approach limits batch size due to its high per-image on-chip storage overhead.

The state-of-the-art designs are too inflexible in their data buffering schemes and control logic to effectively balance between on-chip buffer capacity and off-chip transfer bandwidth. Furthermore, all prior designs consider batch processing only for the fully-connected layers. Because the fully-connected layers typically contain more than 90% of the CNN weights, this yields the most benefit compared to unbatched designs. However, after the weight transfer of fully-connected layers is reduced, the weight transfer of convolutional layers becomes a new bottleneck and calls for batch processing.

III. Analysis of Batching

In this section, we use a simple serial CNN accelerator model to show that the key to efficient batch processing is the trade-off between weight transfer and input transfer, and we show how to choose the best batch size. In Section IV, we extend this idea to construct a practical and scalable CNN accelerator.

A. Modeling Batching CNN Accelerators

Figure 1 shows a model of a simple batch processing CNN accelerator. To simplify analysis, this model has only one multiplier and adder. (We remove this restriction in Section IV.) In each cycle, a value from the input buffer (I) is multiplied with one from the weight buffer (W), and the result is accumulated into the output buffer (O). The system processes a batch of G images at once, so the input and output buffers are evenly divided into G partitions, and data for different images are marked by different colors. The weight buffer is not partitioned because weights are shared by the entire batch.

Batch Processing Fully-Connected Layers. Listing 1 shows the pseudocode for evaluating a fully-connected layer for one image. The y_i loop iterates over all Y output vector values, while the x_i loop iterates over all X input vector values. Each weight $Wt[y_i][x_i]$ is used only once.

When computing fully-connected layers, each cell in Figure 1 (a cell is a square marked with I, O, or W) stores one value. Without batching ($G = 1$), the accelerator reads one input value and reuses it to update all currently buffered outputs, and then advances to the next input. A new weight is read for each output, because weights are not reused within an image. After all inputs are traversed, values in the output buffer are finalized and written to off-chip memory. Computation then moves to the next tile of outputs, until all Y

[1]Here, *batching* refers to buffering data on chip for reuse. This is different from the *mini-batch* used in network training, which controls the gradient descent algorithm.

```
Out[Y]      //output feature vector
In[X]       //input feature vector
Wt[Y][X]    //weight vectors
Bias[Y]     //biases
for(yi = 0; yi < Y; yi++)
  Out[yi] = Bias[yi]
  for(xi = 0; xi < X; xi++)
    Out[yi] += In[xi] * Wt[yi][xi]
```

Listing 1: Pseudocode of an unbatched fully-connected layer.

output values are complete. When a batch comprises more than one image ($G > 1$), computations for different images are interleaved so that each weight is reused G times after being read into the W cell.

Batch Processing Convolutional Layers. A convolutional layer can be viewed as a generalization of a fully-connected layer by replacing each value in the input/output vectors (X and Y) with 2D feature maps and replacing each weight with a 2D kernel. Then, instead of multiplying an input value with a weight value, we convolve a 2D input feature map with a 2D kernel.

For convolutional layers, output feature maps can be partitioned. For example, a layer with output feature maps of size $R \times C$ can be partitioned into $\lceil R/T_r \rceil \times \lceil C/T_c \rceil$ sub-layers with output feature maps of size $T_r \times T_c$. (Sub-layers at boundaries can have smaller feature map sizes). When considering the nested-loop implementation of convolutional layers, this corresponds to tiling the R and C loop with factor T_r and T_c [15].

In summary, when batch processing convolutional layers using the model in Figure 1, an O cell stores $T_r \times T_c$ words, an I cell stores $T_{ri} \times T_{ci}$ words ($T_{ri} = (T_r - 1) \cdot S + K$, and the same for T_{ci}), and a W cell stores $K \times K$ words. Computations for different images are interleaved as in fully-connected layers.

B. Choosing Batch Size for FC Layers

To increase batch size within a given storage budget, one must reduce the data buffered per image. Increasing weight reuse reduces the bandwidth required for weights, but increases transfers of the input data, becoming a critical trade-off for minimizing bandwidth.

Storage per Image vs. Input Transfer. We consider how reducing the storage used by each image affects the amount of input transfer required. In Figure 1, if all Y words are buffered for an image (where Y is the size of the output feature vector), then the inputs only need to be loaded once to compute all Y outputs. This minimizes the input transfer. However, if we store $Y/2$ output words, then the input feature vector must be transferred twice, doubling the input bandwidth required. We define parameter H to control this trade-off. If the output buffer holds Y/H words per image, the processor must read its input H times from off-chip memory. Therefore, $H = 1$ represents the minimum input transfer and $H = Y$ represents the minimum on-chip buffer size.

Batch size G requires an output buffer of size $G \cdot Y/H$. The G input feature vectors (each of size X) must each be transferred H times, incurring a total of $G \cdot XH$ words transferred per batch, and

$$\text{Inputs transferred per image} = XH. \qquad (1)$$

For a fixed storage budget, the input transferred per image grows linearly with the decrease in the amount of buffered output.

Storage Per Image vs. Weight Transfer. Next, we examine the relationship between the storage that an accelerator uses for each image and the amount of weight data that must be transferred per

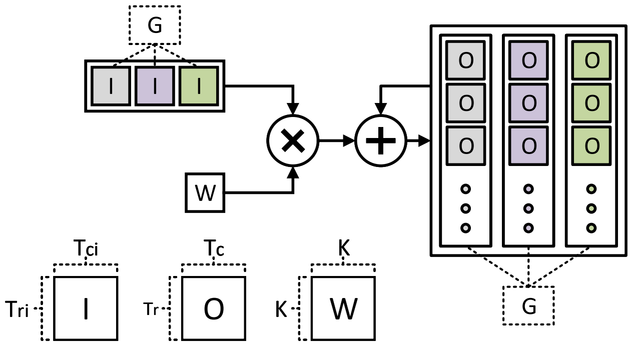

Fig. 1: A simple batch processing CNN accelerator model.

image. We define M_o as the storage budget for the output buffer.[2] Because the output buffer holds $G \cdot Y/H$ words, we require that $G \cdot Y/H \leq M_o$. From this, we can see that the maximum batch size is

$$G = M_o H / Y. \qquad (2)$$

The maximum possible batch size increases linearly with H. Note that (2) allows us to re-write (1) in terms of G: $XH = XYG/M_o$.

Recall that each fully-connected layer uses XY weights. When batching G images, this requires a total of XY/G weights to be transferred per image. Combining this with (2),

$$\text{Weights transferred per image} = \frac{XY}{G} = \frac{XY^2}{M_o H}. \qquad (3)$$

X and Y are constants dictated by the layer's dimensions. Therefore, for a fixed storage budget, the number of weights transferred per image scales with $1/G$ (or equivalently, $1/H$).

Trade-off: Input vs. Weight Transfer. Equations (1) and (3) describe a trade-off: given a fixed on-chip storage budget, reducing the amount of output data buffered per image allows increasing the batch size G, scaling the per-image input transfer up by a factor of G, while scaling the per-image weight transfer down by G.

Figure 2 illustrates this behavior by showing the total number of inputs and weights transferred per image (summing (1) and (3), while expressing (1) in terms of G: $XH = XYG/M_o$). We examine the first fully-connected layer of VGGNet-E for four different storage budgets, assuming each word is 32 bits. Note that, when $G = 1$ (no batching used), the y-axis goes to 392MB per image; we truncate the y-axis for readability.

When G is small, increasing it is extremely beneficial. On the left side of the graph, weight transfer XY/G (3) dominates; increasing G dramatically reduces the data transferred. However, as G increases, all storage curves reach an inflection point, where the input transfer XYG/M_o (1) begins to dominate. Beyond these points, input transfer outweighs the benefits of increasing G.

These results have an interesting implication on how to choose the best batch size. For example, storing all Y outputs for each image and setting the batch size to be $G = M_o/Y$ is sub-optimal, because it limits the batch size to stay to the left of the inflection point. Alternatively, using the largest batch size allowed within the accelerator's latency constraint is also likely sub-optimal, because this may increase the batch size to be on the right of the inflection point.

Analytically, assuming continuous variables, one can show that the optimal batch size (which minimizes the sum of (1) and (3))

[2]We focus on the output storage M_o because the buffers required for inputs and weights are much smaller, only $1+G$ words in fully-connected layers.

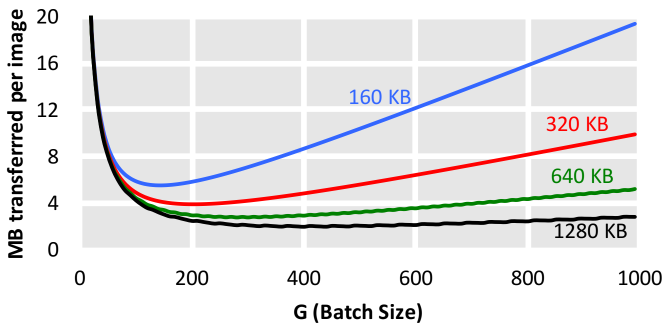

Fig. 2: Sum of input and weight transferred per image vs. batch size G, for different storage budgets M_o, for the first FC layer of VGGNet-E.

is $G = \sqrt{M_o}$. However, actual designs are discrete in nature and require multiple buffer banks for parallel computation. (One may also need to limit the maximum batch size due to its effect on the overall latency of the system.) So, rather than setting $G = \sqrt{M_o}$, we use an exhaustive search to determine the best value of G.

Interpretation as Matrix-Matrix Multiplication. Another way to understand batching of fully-connected layers is to think of it as a matrix-matrix multiplication. In this case, the two matrices are M_W and M_I, where M_W is the weight matrix, with Y rows and X columns, and M_I is the input matrix, with X rows and as many columns as there are possible input images (typically unbounded). Batching can then be viewed as a blocking problem, partitioning M_W into blocks of Y/H rows and M_I into blocks of G columns. Viewed this way, the trade-off we identify between the bandwidth of weights and input data becomes equivalent to the trade-off found when blocking the M_W and M_I matrices [16].

C. *Batching for Convolutional Layers*

Because we can view a convolutional layer as a generalization of an FC layer (see Section III-A), the trade-off between input transfer and weight transfer observed for fully-connected layers also exists for convolutional layers. That is, in an FC layer, each of the *I*, *O*, and *W* cells in Figure 1 consists of a single value, whereas in a convolutional layer, each cell is a 2D structure. As a result, batch processing of convolutional layers is analogous to the fully-connected layers: a batch of G independent inputs share weights, amortizing the weight transfer bandwidth across the inputs.

However, because multiply-add operations in fully-connected layers are generalized to 2D-convolution operations in convolutional layers, the details of how G affects input and weight transfer in convolutional layers are more complicated than the case of fully-connected layers. To be specific, two more parameters must be considered: T_r and T_c dictate the tiling of the convolutional layer's two-dimensional inputs and outputs. These parameters affect the on-chip storage, input transfer, and weight transfer requirements [15]. For convolutional layers, the goal of our optimization is not merely to find the best value of G (as for FC layers), but to find the best set of (T_r,T_c,G). Given a layer's parameters and memory budget, we use an exhaustive search to find the best combination.

IV. Escher CNN Accelerator

In Section III, we demonstrated the importance of controlling the batch size and the amount of data stored per image, using a simple accelerator model with only one multiplier and one adder as compute units. We now extend this idea to design a practical CNN accelerator that uses an array of vector dot product units and adders to exploit parallelism in CNN computations, while retaining the flexibility in data buffering to enable the accelerator to balance input transfer and weight transfer. Similar to Section III, we first consider only fully-connected layers and then generalize to convolutional layers.

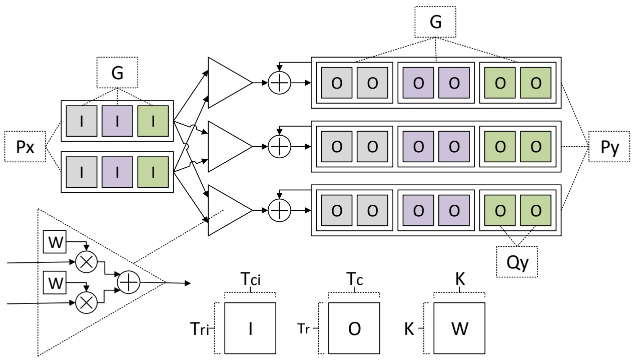

Fig. 3: Structure of an Escher CNN accelerator.

A. *Practical FCLP Design*

Figure 3 presents our Escher FCLP (Fully-Connected Layer Processor) design. Analogous to prior work [17], this design exploits two forms of parallelism. First, multiple values from a single input feature vector can be accumulated into one output value using a parallel vector dot product unit (input parallelism). P_x is used to represent the number of inputs that each vector dot product unit takes from a single input feature vector. Second, multiple output values can be computed in parallel by using multiple parallel vector dot product units (output parallelism). P_y is used to represent the number of parallel vector dot product units. We note that, to provide sufficient access bandwidth, the organization of the FCLP's on-chip buffers must match the data access pattern of its compute units: the input buffers are split into P_x banks, the output buffers are split into P_y banks, and the weight buffer is partitioned into $P_x \times P_y$ banks.

Listing 2 shows the pseudocode corresponding to Figure 3. There are four parameters: G (batch size), P_y (output parallelism), P_x (input parallelism), and Q_y, where $Q_y \times P_y$ outputs are buffered for each image. Out, In, Wt, and $Bias$ are in off-chip memory, while o_{buf}, i_{buf}, and w_{buf} are on-chip buffers. All on-chip storage uses double-buffering to overlap computation with data transfer. The unrolled p_y and p_x loops correspond to the parallel compute units. Parameter H is computed based on the analysis presented in Section III-B. Note that the off-chip data array declarations are written for readability in the pseudocode; an actual implementation may use a different layout to optimize for DRAM access locality.

B. *Optimizing an FCLP Design*

For a given fully-connected layer, the parameters in Listing 2 are optimized to find the best performance (throughput) within a set of resource constraints (i.e., bandwidth, on-chip storage, and multiplier and adder units). The following formulas are used to estimate the characteristics of the candidate designs during optimization:

- Throughput is derived from the clock frequency and cycle count; it takes $S_y \cdot S_x \cdot Q_y \cdot G$ clock cycles to process one batch of G input vectors.

- Computation resources are based on the template structure, which uses $P_y \times P_x$ multipliers and adders.
- On-chip storage for the input buffers holds $G \cdot P_x$ words, but requires $G \cdot P_x \cdot 2$ words because it is double buffered.
- On-chip storage for the output buffers requires $G \cdot Q_y \cdot P_y \cdot 2$ words and the weight buffers require $P_x \cdot P_y \cdot 2$ words.
- Memory bandwidth is computed from the data transferred per batch of images and the design throughput. For one batch G, the system loads the input buffers $S_y \cdot S_x$ times, each time reading $G \cdot P_x$ words. The output buffers are loaded and stored S_y times, with $Q_y \cdot P_y$ words loaded and $G \cdot Q_y \cdot P_y$ words stored. The weight buffers are loaded $S_y \cdot S_x \cdot Q_y$ times, reading $P_y \times P_x$ words each time.

The goal of the optimization is to find the (G, Q_y, P_y, P_x) combination with the highest throughput within the given resource limit. The set of possibilities is small enough that the best option can be easily found through an exhaustive search. It is also possible to jointly optimize an FCLP for multiple fully-connected layers. In this case, the same (P_y, P_x) are used for all layers because they dictate the number and organization of the compute units and memories, but Q_y and G can change from layer to layer because changing them only affects the runtime-adjustable control logic parameters. When jointly optimizing for multiple fully-connected layers, the buffers are sized for the layer with the largest storage demand, typically the first fully-connected layer.

C. Integration with Convolutional Layers

We extend the fully-connected design in Figure 3 to convolutional layers by viewing them as a generalization of the fully-connected layers (see Section III-C). When processing fully-connected layers, each O cell stores one value, but when processing convolutional layers, each O cell stores a feature map of size $T_r \times T_c$. Similarly, for the fully-connected layers, an I cell stores one value, while convolutional layers store a feature map of size $[(T_r - 1) \times S + K] \times [(T_c - 1) \times S + K]$, where S is the stride and K is the filter kernel size. Therefore, the fundamental structure of the computation remains the same as for the fully-connected layers, but adds two more loops (T_r and T_c) between the G and P_y loops in Listing 2. Furthermore, for the convolutional layers, each W cell also stores a filter of size $K \times K$. Similarly, this adds two K loops between the G and P_y loops in Listing 2. The resulting design is sufficient to process convolutional layers, and can be used to process fully-connected layers by setting T_r*=1*, T_c*=1*, and *K=1*. Note that, when *G=1* and Q_y*=1*, the convolutional layers are processed without batching, which is equivalent to [15].

D. Optimizing for Convolutional and Fully-Connected Layers

To simultaneously optimize an Escher accelerator design for convolutional and fully-connected layers, we extend the optimization model to cover the convolutional layers. When processing a convolutional layer, a multiplication is generalized to a 2D convolution and will take $T_r \cdot T_c \cdot K^2$ cycles to complete. The convolutional layer is partitioned into $\frac{R_{out}}{T_r} \times \frac{C_{out}}{T_c}$ sub-layers. Thus, the number of cycles needed to compute a batch of images is $S_y \cdot S_x \cdot Q_y \cdot G \cdot R_{out} \cdot C_{out} \cdot K^2$.

For on-chip storage, compared to the FC layer formulas, the input buffer is scaled by $[(T_r - 1) \times S + K] \times [(T_c - 1) \times S + K]$,

```
// parameters
G, Qy, Py, Px
// derived parameters
H = Y/(Qy*Py), Sy = ceil(H), Sx = ceil(X/Px)
// off chip
Out[G][Sy][Qy][Py]      // contains Out[G][Y]
In[G][Sx][Px]           // contains In[G][X]
Wt[Sy][Sx][Qy][Py][Px]  // contains Wt[Y][X]
Bias[Sy][Qy][Py]        // contains Bias[Y]
// on chip
obuf[G][Qy][Py], ibuf[G][Px], wbuf[Py][Px]
for(sy = 0; sy < Sy; sy++)
  obuf[0:G-1] = repeat(Bias[sy],G)
  for(sx = 0; sx < Sx; sx++)
    ibuf[0:G-1] = In[0:G-1][sx]
    for(qy = 0; qy < Qy; qy++)
      wbuf = Wt[sy][sx][qy]
      for(g = 0; g < G; g++)
        // py,px unrolled for parallelism
        for (py = 0; py < Py; py++)
          for (px = 0; px < Px; px++)
            obuf[g][qy][py] +=
              ibuf[g][px] * wbuf[py][px]
  Out[0:G-1][sy] = obuf[0:G-1]
```

Listing 2: Pseudocode of a fully-connected Escher layer.

the weight buffer by $K \times K$, and the output buffer by $T_r \times T_c$. For data transfer, first, the same scaling is applied as for the storage, and then the result is multiplied by $\frac{R_{out}}{T_r} \times \frac{C_{out}}{T_c}$ to account for looping through the sub-layers.

Finally, when optimizing for the convolutional layers, T_r and T_c must be optimized for each layer, because the best (T_r, T_c) may be different for different convolutional layers.

V. Evaluation

We compare Escher with several prior works. We generate accelerator designs for AlexNet, VGGNet-E, and GoogleNet that use 32-bit floating point or 16-bit fixed point data types. All accelerator designs are optimized targeting 60% utilization of the available DSPs and BRAMs of a Virtex-7 690T FPGA at 100 MHz.
Optimization. All designs are optimized to accommodate all convolutional and fully-connected layers in the target CNN. For each design, we first choose P_y and P_x to maximize throughput, then choose the remaining parameters to minimize the peak bandwidth. If there are multiple (P_y, P_x) with the same throughput, the design with minimum peak bandwidth is selected. If multiple designs have the same peak bandwidth requirements, the design with the lower average bandwidth is selected. This strategy minimizes the bandwidth of all CNN layers without harming performance. To limit optimization runtime, we do not consider batch sizes greater than 300, as they offer negligible bandwidth benefits.
RTL. We used Vivado 2016.4 to generate RTL for all accelerator designs from C++ source code decorated with HLS #pragma directives. For 32-bit floating-point designs, the adders used for accumulation are pipelined; we require that $G \cdot Q_y \cdot T_r \cdot T_c$ is larger than the adder's pipeline depth to ensure that loop-carry dependencies are avoided.
Prior Art. We compare Escher to the Caffeine [11] input-major (*Caffeine-I*) and weight-major (*Caffeine-W*) designs. We also compare Escher to the batching strategy from [7] (*Fudan*) by setting Q_y to $\lceil Y/P_y \rceil$, which corresponds to buffering the maximum amount of data per image. To highlight the benefits of batching of the convolutional layers, we also compare with a restricted Escher

TABLE I: Peak bandwidth, in GB/s.

	32-bit Floating Point			16-bit fixed point		
	AlexNet	VGG-E	Google	AlexNet	VGG-E	Google
Escher	0.90	1.40	4.37	2.05	1.71	12.03
Escher-FConly	1.26	1.50	5.31	3.36	2.89	14.20
Fudan	2.09	1.57	5.31	4.97	2.89	14.20
Caffeine-I	2.81	5.40	5.31	7.35	7.35	14.20
Caffeine-W	2.43	5.03	5.31	6.45	6.45	14.20

TABLE II: Throughput, in Gops/s.

	32-bit Floating Point			16-bit fixed point		
	AlexNet	VGG-E	Google	AlexNet	VGG-E	Google
(all designs)	62	81	67	135	393	224

design that batches only the fully-connected layers (*Escher-FConly*) by setting *G=1* and Q_y*=1* for the convolutional layers, which mimics how Caffeine handles these layers. In all four designs, where convolutional layer processing is not batched, these layers are optimized based on [15].

A. Peak Bandwidth Comparison

In Section III, we demonstrated the bandwidth benefits of Escher for the fully-connected layers using an analytical model. We now compare an actual Escher accelerator design with accelerators built using other methods. When processing a CNN layer, bandwidth use is relatively constant. However, across layers, it can vary greatly. Therefore, we define the *peak bandwidth* of a design as the bandwidth that ensures the accelerator is not bottlenecked by data transfer when processing the most bandwidth-demanding CNN layer.

Table I shows the peak bandwidth requirements of each design. Assuming compute is not blocked by data transfer, for the same CNN and data type, the five different methods produce accelerators with less than one percent throughput difference, making their bandwidth directly comparable. For reference, Table II shows the common throughputs when peak bandwidth requirements are satisfied.

First, we examine the benefits of Escher when applied only to the fully-connected layers (*Escher-FConly*). Compared to the best among the state of the art (*Fudan*), *Escher-FConly* reduces peak bandwidth by up to $1.7\times$ (*AlexNet floating point*). We then compare *Escher-FConly* with *Escher*, which shows that batching for convolutional layers further reduces peak bandwidth by up to another $1.7\times$ (*VGG fixed point*). Notably, *Escher-FConly* shows no benefit for GoogleNet, while *Escher* is able to reduce peak bandwidth. This happens because the GoogleNet designs are bottlenecked by the data transfer on the convolutional layers.

Overall, Escher consistently lowers peak bandwidth. When both convolutional and fully-connected layers are batched, Escher reduces peak bandwidth by up to $2.4\times$ (*AlexNet fixed point*) compared to *Fudan* and even more compared to the *Caffeine* designs.

B. Per-layer Bandwidth Comparison

A clearer picture of how the different design methods perform can be observed by examining the bandwidth requirements of the individual layers. Due to the large sizes of VGGNet-E and GoogleNet (VGGNet-E has 16 convolutional layers and 3 fully-connected layers; GoogleNet has 57 convolutional layers and 1 fully-connected layer [18]), we only present a representative sample of layers for these networks. Specifically, we include all fully-connected layers, the bottleneck layers for all designs, and the convolutional layers in which Escher shows the greatest and least advantages.

TABLE III: Per-layer bandwidth breakdown, 16-bit fixed point, in GB/s.

	Escher	Escher-FConly	Fudan	Caffeine-I	Caffeine-W
AlexNet					
Conv. 1a/1b	0.54	0.29	3.45	0.28	0.28
Conv. 2a/2b	0.55	0.87	1.71	0.87	0.87
Conv. 3a/3b	0.74	3.30	3.30	3.30	3.30
Conv. 4a/4b	0.80	3.36	3.36	3.36	3.36
Conv. 5a/5b	0.87	**3.36**	3.36	3.36	3.36
FC 1	2.00	2.66	4.92	7.29	6.39
FC 2	**2.05**	2.72	**4.97**	**7.35**	6.44
FC 3	1.92	2.78	1.71	7.29	**6.45**
VGGNet-E					
Conv. 2	1.46	1.40	1.40	1.40	1.40
Conv. 13–16	0.44	**2.89**	**2.89**	2.89	2.89
FC 1	1.59	1.59	2.76	7.27	6.07
FC 2	1.67	1.67	2.84	**7.35**	6.15
FC 3	**1.71**	1.71	1.71	7.29	**6.45**
GoogleNet					
Conv. 1	0.39	0.37	0.37	0.37	0.39
Conv. 2	**12.03**	12.04	12.04	12.04	12.04
Conv. 46	2.32	**14.20**	**14.20**	**14.20**	**14.20**
Conv. 54	0.87	9.10	9.10	9.10	9.10
FC 1	2.19	2.35	2.40	7.57	6.74

Table III shows the per-layer bandwidth breakdowns, with the peak bandwidth for each design shown in bold. We make three observations. First, batching can reduce convolutional layer bandwidth by far more than the reduction in peak bandwidth (e.g., $6.6\times$ improvement on *VGGNet-E Conv. 13–16*, compared to $1.7\times$ improvement to the peak bandwidth). Although these savings do not contribute to throughput, they save energy and reduce the overall data transfer.

Second, in some non-bottleneck layers, Escher may increase the bandwidth compared to the other techniques. Because our optimization goal is to minimize peak bandwidth requirements for the entire CNN, we may prioritize the bottleneck layer(s) at the occasional expense of increasing bandwidth in other layers. In particular, convolutional layers have greater demand for large input buffers compared to the fully-connected layers, because a single input value in a fully-connected layer corresponds to a 2D feature map of values in a convolutional layer. This effect is particularly significant for early-stage convolutional layers because they have larger feature maps.

Third, without support for batching in convolutional layers, an accelerator may be bottlenecked by the convolutional layer bandwidth requirements, as evidenced by Escher-FConly, which is limited by convolutional layers in all three networks.

GoogleNet highlights an interesting scenario, where the bandwidth bottleneck for all designs is a convolutional layer. Escher is bottlenecked by Conv. 2, whereas all other designs are bottlenecked by Conv. 46. This happens because Escher successfully reduces the bandwidth requirements of Conv. 46 through batching. However, batching is ineffective for Conv. 2, because the early convolutional layers have very few weights [19], thus reusing them makes little difference. Notably, the largest bandwidth reduction in GoogleNet is still observed in a convolutional layer, with Escher using $10.5\times$ less bandwidth than the other designs on Conv. 54.

C. FPGA Resource Comparison

To illustrate that Escher does not incur an FPGA resource overhead, we present the post place-and-route resource consumption

TABLE IV: FPGA resource utilization.

		BRAM-18K	DSP	FF	LUT
32-bit floating point					
AlexNet	Escher	1,765(60%)	2,225(62%)	179K(21%)	181K(42%)
	Caffeine	1,645(56%)	2,224(62%)	180K(21%)	181K(42%)
	Fudan	1,765(60%)	2,224(62%)	181K(21%)	184K(43%)
VGGNet-E	Escher	1,765(60%)	2,213(61%)	213K(25%)	187K(43%)
	Caffeine	1,591(54%)	2,213(61%)	215K(25%)	187K(43%)
	Fudan	1,743(59%)	2,212(61%)	214K(25%)	188K(43%)
GoogleNet	Escher	1,753(60%)	2,226(62%)	181K(21%)	178K(41%)
	Caffeine	1,765(60%)	2,225(62%)	183K(21%)	184K(42%)
	Fudan	1,765(60%)	2,225(62%)	183K(21%)	184K(42%)
16-bit fixed point					
AlexNet	Escher	1,745(59%)	2,182(61%)	125K(14%)	112K(26%)
	Caffeine	1,765(60%)	2,182(61%)	129K(15%)	113K(26%)
	Fudan	1,779(61%)	2,181(61%)	129K(15%)	118K(27%)

in Table IV (we place-and-route the CNN accelerators in isolation, omitting memory controllers, etc.). All designs meet post place-and-route timing at 100MHz using Vivado 2016.4. The resources for Caffeine input-major and weight-major are nearly identical; therefore, we report only the lesser result of these two designs.

The key takeaway is that Escher has effectively the same resource utilization as the state-of-the-art designs, achieving the same or better performance and significantly reducing the off-chip bandwidth utilization. This occurs because the vast majority of the FPGA resources in all designs are dedicated to the datapath, whose resource consumption is dictated by the optimization target (60% DSP and 60% BRAM in our results).

The only noticeable resource utilization difference for Escher is a marginally higher BRAM count compared to Caffeine on the AlexNet and VGGNet-E floating-point designs. This happens because, despite being allowed to use up to 60% of the BRAMs, the Caffeine designs do not benefit from the extra storage capacity, yielding designs that use only 54%–56% of the BRAMs. Escher can effectively utilize the entire BRAM budget to reduce bandwidth use.

D. *Sensitivity to Latency Constraints*

Although batching is required to make off-chip bandwidth requirements for CNN accelerators practical for today's FPGAs, it necessarily increases the per-image processing latency. In latency-sensitive environments, the maximum size of a batch may be limited by the latency target, which takes precedence over minimizing bandwidth consumption.

Figure 4 presents the peak bandwidth of the 16-bit fixed-point AlexNet designs as a function of the maximum batch size allowed in the optimization. The batch size used by each layer is independently chosen to minimize the overall peak bandwidth, with the only additional constraint to not exceed the maximum allowed batch size. All designs we present have negligible throughput differences, enabling direct comparison of their peak bandwidth.

The Caffeine weight-major design (*Caffeine-W*) constrains the batch size to match the number of vector dot product units (here, $P_y = 66$), making this design inapplicable when the maximum permissible batch size is lower than 66. Moreover, this design cannot benefit from larger batches (indicated by a flat line starting from 66). For small batch sizes, Escher and Fudan overlap, because both designs will store complete output vectors on chip. However, the Fudan technique does not scale to larger batch sizes because it is

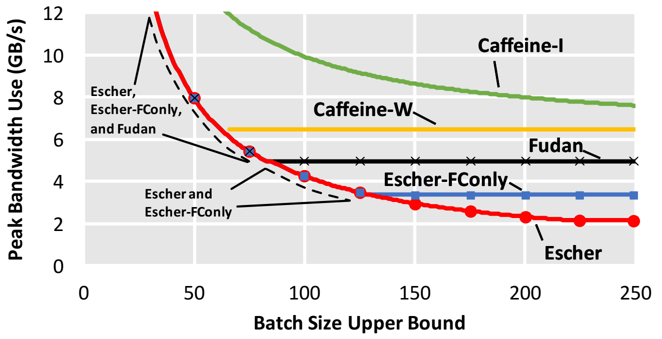

Fig. 4: Max batch size vs. peak bandwidth for AlexNet 16-bit fixed point.

unable to reduce the amount of buffered data per image (indicated by a flat line beyond 83). Escher-FConly overlaps with Escher when convolutional layers are not the bottleneck, which is the case when the batch size is <130. Beyond that point, the convolutional layers become the bottleneck, and Escher-FConly also flattens out.

Only the Caffeine input-major (*Caffeine-I*) and Escher designs continue to benefit from larger batch sizes. The optimization process can choose designs with higher latencies (larger batches) to reduce the peak bandwidth requirements. Eventually, both designs reach diminishing returns, beyond which the bandwidth benefits of further increasing the batch size are negligible. However, Caffeine input-major has a fixed overhead, re-transferring inputs $\lceil Y/P_y \rceil = \lceil 4096/66 \rceil = 62$ times for the bottlenecked fully-connected layer. Importantly, for all batch sizes, Escher has the lowest peak bandwidth requirements.

VI. Related Work

In addition to the designs we evaluate [7], [11], batch processing of the fully-connected layers is mentioned in passing in [10]; however, little detail is provided.

The memory bandwidth problem faced by CNN accelerators has been studied in previous work. [13] uses data quantization to reduce the word size for both convolutional layers and fully-connected layers, and further uses SVD decomposition to compress the weights in fully-connected layers. [20] uses the DeepCompression method [21] to compress weights in fully-connected layers. A limitation of compression-based techniques is that they cannot be used during training. [15], [17], [22] use loop tiling to reduce the bandwidth requirements of convolutional and/or fully-connected layers, but without batching. To avoid off-chip data transfer, [9] and [23] consider designs that store all weight data on chip. For [9], eDRAM is used, while [23] targets small CNNs. In both works, the on-chip storage limits the supported CNN size. [7] buffers all input/output data on chip, and similarly limits the supported CNN size. [19] proposes a layer fusion technique that reduces the input/output bandwidth of early stage convolutional layers and is complimentary to Escher.

Beyond memory bandwidth, the compute unit is an important aspect of CNN accelerator design. [24]–[28] use various dataflows that are optimized for performing 2D-convolutions. While highly efficient for convolutional layers, these designs are inefficient for the fully-connected layers. [10], [23] use NoCs with FMA compute nodes. Such designs, used for the flexibility of their compute fabric, mainly target ASIC implementations. Like our work, [9], [11],

[15], [17], [29]–[31] use vector dot product arrays. Dot product units are a natural fit for fully-connected layers because each output is computed by a vector dot product, although they can lead to underutilization problems in convolutional layers due to dimension mismatch. [29], [30] address this problem by grouping convolutional layers based on the compatibility of their dimensions. [7] avoids this problem by mapping each layer to a dedicated compute module.

VII. Conclusions

CNN accelerators that independently process images are bottlenecked by weight data transfer, particularly when processing the fully-connected layers. This prompted a trend in recent CNN accelerators to process images in batches when computing fully-connected layers, enabling the reuse of weights once they are brought on chip. However, previous schemes do not fully leverage batch processing because of the inherent limitations in their accelerator designs, which limit the choice of batch sizes.

In this work, we observed the possibility of trading off weight-transfer bandwidth for input-transfer bandwidth by simultaneously controlling the image batch size and data buffered per image. Leveraging our observations, we developed Escher, a CNN accelerator with high flexibility in selecting the batch size and per-image storage. Moreover, unlike prior batching approaches, Escher is able to save bandwidth by batch processing the convolutional layers in addition to the fully-connected layers. We validated our design methodology by building accelerators for AlexNet, VGGNet-E, and GoogleNet on a Virtex-7 690T FPGA. Escher achieved $2.4\times$ peak bandwidth reduction compared to prior methods on fixed-point AlexNet. On fixed-point GoogleNet, the bandwidth requirements of convolutional layers were reduced by up to $10.5\times$.

Acknowledgment

The authors would like to thank Cheng-Yang Fu and Alex C. Berg from the Computer Vision Group at the University of North Carolina at Chapel Hill for their help. This material is based on work and equipment supported by the National Science Foundation under Grant Nos. 1533739, 1453460 and 1405641.

References

[1] A. van den Oord, S. Dieleman, and B. Schrauwen, "Deep content-based music recommendation," in *27th Conference on Neural Information Processing Systems*, 2013.

[2] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *25th International Conference on Machine Learning*, 2008.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *26th Conference on Neural Information Processing Systems*, 2012.

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

[5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *28th Conference on Computer Vision and Pattern Recognition*.

[6] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," *arXiv:1410.0759*, 2014.

[7] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural network," in *26th Intl. Conference on Field-Programmable Logic and Applications*, 2016.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv:1603.04467*, 2016.

[9] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *47th IEEE/ACM International Symposium on Microarchitecture*, 2014.

[10] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *43rd International Symposium on Computer Architecture*, 2016.

[11] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks," in *35th International Conference on Computer-Aided Design*.

[12] Y. Ma, N. Suda, J.-s. Seo, Y. Cao, and S. Vrudhula, "Scalable and modularized RTL compilation of convolutional neural networks onto FPGA," in *26th Intl. Conference on Field-Programmable Logic and Applications*, 2016.

[13] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *24th International Symposium on Field-Programmable Gate Arrays*, 2016.

[14] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *24th International Symposium on Field-Programmable Gate Arrays*, 2016.

[15] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *23rd International Symposium on Field-Programmable Gate Arrays*, 2015.

[16] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. on Mathematical Software*, vol. 34, no. 3.

[17] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[18] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv:1408.5093*, 2014.

[19] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *49th IEEE/ACM International Symposium on Microarchitecture*, 2016.

[20] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *43rd International Symposium on Computer Architecture*, 2016.

[21] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding," *CoRR, abs/1510.00149*, vol. 2, 2015.

[22] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *31st International Conference on Computer Design*, 2013.

[23] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting vision processing closer to the sensor," in *42nd International Symposium on Computer Architecture*, 2015.

[24] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *37th International Symposium on Computer Architecture*, 2010.

[25] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *International Symposium on Circuits and Systems*, 2010.

[26] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for convolutional networks," in *19th Intl. Conference on Field-Programmable Logic and Applications*, 2009.

[27] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *40th International Symposium on Computer Architecture*, 2013.

[28] 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 *20th Application-specific Systems, Architectures, and Processors*, 2009.

[29] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *44th International Symposium on Computer Architecture*, 2017.

[30] Y. Shen, M. Ferdman, and P. Milder, "Overcoming resource underutilization in spatial CNN accelerators," in *26th Intl. Conference on Field-Programmable Logic and Applications*, 2016.

[31] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li, "C-brain: a deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization," in *53rd Design Automation Conference*, 2016.