Fused-Layer CNN Accelerators

Manoj Alwani
Stony Brook University
malwani@cs.stonybrook.edu

Han Chen
Stony Brook University
han.chen.2@stonybrook.edu

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

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

***Abstract*—Deep convolutional neural networks (CNNs) are rapidly becoming the dominant approach to computer vision and a major component of many other pervasive machine learning tasks, such as speech recognition, natural language processing, and fraud detection. As a result, accelerators for efficiently evaluating CNNs are rapidly growing in popularity. The conventional approaches to designing such CNN accelerators is to focus on creating accelerators to iteratively process the CNN layers. However, by processing each layer to completion, the accelerator designs must use off-chip memory to store intermediate data between layers, because the intermediate data are too large to fit on chip.**

In this work, we observe that a previously unexplored dimension exists in the design space of CNN accelerators that focuses on the dataflow *across* convolutional layers. We find that we are able to *fuse* the processing of multiple CNN layers by modifying the order in which the input data are brought on chip, enabling caching of intermediate data between the evaluation of adjacent CNN layers. We demonstrate the effectiveness of our approach by constructing a fused-layer CNN accelerator for the first five convolutional layers of the VGGNet-E network and comparing it to the state-of-the-art accelerator implemented on a Xilinx Virtex-7 FPGA. We find that, by using 362KB of on-chip storage, our fused-layer accelerator minimizes off-chip feature map data transfer, reducing the total transfer by 95%, from 77MB down to 3.6MB per image.

I. Introduction

Deep convolutional neural networks (CNNs) have revolutionized the accuracy of recognition in computer vision. More broadly, this is part of a trend—using deep neural networks with many layers—that has been instrumental to rapid progress in many diverse fields, including natural language processing, information retrieval, computational biology, and speech recognition.

Underlying the accuracy improvements of CNNs are massive increases in computation. With each newly developed network, the accuracy of recognition increases and the number of computations required to evaluate the network grows. Already, general-purpose CPUs have become a limiter for modern CNNs because of the lack of computational parallelism. As a result, there has been significant interest in developing and adapting hardware accelerators for CNNs [7] such as GPUs [3], FPGAs [13], [19], [10], [1], and ASICs [2].

Although the CNN computation is mathematically simple, the sheer volume of operations precludes a dataflow implementation even for a single layer. Each convolution layer requires iterative use of the available compute units. Research into the design of CNN accelerators has therefore concentrated on developing a CNN building block that iteratively evaluates the network. A number of methodologies have been developed for optimizing the architecture of such CNN accelerator building blocks, concentrating either on specific constraints [10] or evaluating the design space of compute units and memory bandwidth [19].

Traditional implementations of CNNs (both hardware and software) evaluate the network by following its structure, one layer at a time. This approach produces a large amount of intermediate data that are gradually streamed out to memory as the computation progresses. Upon completing a layer, the intermediate data are streamed back to the same compute units, repeating the process until all layers have been evaluated. As CNNs grow larger, the amount of intermediate data that must be shuttled between the compute units and memory increases.

We observe that an additional, previously unexplored, dimension exists for CNN accelerator architectures that focuses on the dataflow *across* convolutional layers. Rather than processing each CNN layer to completion before proceeding to the next layer, it is possible to restructure the computation such that multiple convolutional layers are computed together as the input is brought on chip, obviating the need to store or retrieve the intermediate data from off-chip memory. This accelerator organization is made possible by the nature of CNNs. That is, each point in a hidden layer of the network depends on a well-defined region of the initial input to the network.

In this work, we develop a novel CNN evaluation strategy that breaks away from the commonly accepted practice. By modifying the order in which the original input data are brought on chip, changing it to a pyramid-shaped multi-layer sliding window, our architecture enables effective on-chip caching during CNN evaluation. The caching in turn drastically reduces the off-chip memory bandwidth requirements, minimizing data movement.

We validate our approach by demonstrating CNN layer fusion on a Xilinx Virtex-7 FPGA. Using analytic modeling and our FPGA prototype, we demonstrate the ability to restructure the CNN evaluation such that intermediate data do not need to be shuffled on and off chip. In our evaluation of the first five convolutional layers of VGGNet-E, our method requires only 362KB of storage to replace 73MB of off-chip feature map transfers. Additionally, we created a mathematical model for reasoning about and searching over the layer fusion design space and evaluating its tradeoffs. Lastly, we describe and provide pseudo-code for a high-level synthesis template for implementing our design.

978-1-5090-3508-3/16/$31.00

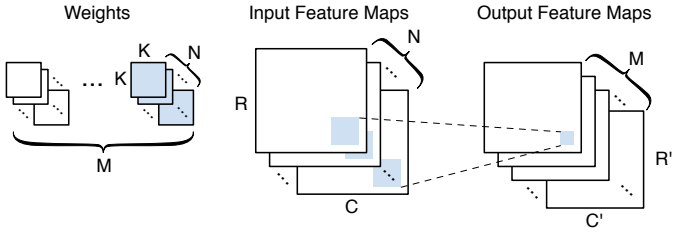


Fig. 1. Illustration of a convolutional layer.

The rest of this paper is organized as follows. In Section II, we introduce CNNs and CNN hardware accelerators. Section III introduces our new method for fusing multiple convolutional layers, eliminating off-chip data movement between them. Section IV describes our architecture and its implementation using high-level synthesis. Section V explores the fused-layer tradeoff space and Section VI presents an FPGA-based evaluation of our designs. Section VII discusses related work and Section VIII concludes.

II. Designing CNN Accelerators

A convolutional neural network (CNN) performs feature extraction using a series of *convolutional layers*, typically followed by one or more dense ("fully connected") neural network layers that perform classification. Figure 1 shows an example of one convolutional layer. Each layer takes as input a set of N *feature maps* (N channels of $R \times C$ values), and convolves it with M sets of $N \times K \times K$ filters (whose weights were previously determined through a learning algorithm such as back propagation). For each of the M sets, the convolution is performed by sliding the filter across the input feature map with a stride of S (where a filter moves S locations at each step). At each location, a filter's values are multiplied with the overlapping values of the input feature maps. The resulting products are summed together, giving one value in an output feature map. This process is repeated for each of the M filter sets, with each repetition giving one output feature map. (Each of the M output feature maps has dimensions $R' \times C'$, where $R' = \frac{R}{S} - \frac{K}{S} + 1$ and $C' = \frac{C}{S} - \frac{K}{S} + 1$). Additionally, one bias value (also determined through back propagation) is added to each of the M different output feature maps. Typically, the output feature maps then undergo a non-linear operation (e.g., ReLU [8]), optionally followed by a subsampling operation (e.g., pooling). The nonlinear and subsampling operations are small, working locally on a single channel of the feature map; these typically consume a very small percentage of the overall computation.

Convolutional networks comprise many layers, with the outputs of the preceding layer being used as the input feature maps of the subsequent layer. Increasing the depth (number of layers) of a network yield higher recognition accuracy [17]. Because of this, the past few years have shown a marked increase in the number of layers used in state-of-the art CNNs. For example, AlexNet [8], the winner of the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [15], uses five convolutional layers, while VGG [17] (winner of a portion of ILSVRC 2014) uses up to 17 convolutional layers. Meanwhile, GoogLeNet [18] (which also won a portion of the 2014 ILSVRC) uses 20 convolutional layers, using kernels ($K \times K$) as small as 1×1 to allow an increased network depth to be computationally feasible.

In this paper, we focus on the convolutional layers (as well as the subsampling layers that typically surround them), and not on the final fully connected layers, which perform a dense neural network operation. Fully connected layers have lower computational cost than convolutional layers and their data usage is dominated by the layer's weights, not by the input and output feature maps.

A. Hardware Accelerators

The computational dominance of the convolutional layers, coupled with the need for larger and deeper networks, has sparked significant interest in the design and optimization of accelerator structures for these layers [19], [10]. These techniques construct accelerators consisting of the multipliers and adders needed to perform the convolution, as well as the on-chip memory buffers to hold data and filter weights. The accelerators are used iteratively, performing one layer of computation at a time. For each layer, input feature maps and filter weights are brought from off-chip DRAM into local buffers, the convolutions are performed, and output feature map data are written into DRAM. The large volume of data comprising the feature maps stresses the memory system and can become the bottleneck. This has inspired efforts to optimize the memory accesses patterns for this layer-by-layer approach. For example, [10] and [19] use loop transformations with the goal of balancing resources between arithmetic structures, on-chip memories, and memory bandwidth.

B. Data Access Patterns

We observe that, although a number of approaches have focused on effectively managing on-chip memory and off-chip bandwidth while evaluating a convolutional layer, existing approaches forgo the possibility of restructuring the computation *across* layers to minimize bandwidth usage. Because prior approaches consider each convolutional layer separately, they start with the assumption that every layer must bring the input feature maps from off-chip DRAM and must write the output feature maps back when the computation finishes. This transfer of feature map data to and from external memory is costly in terms of memory bandwidth and energy [10]. As deep learning algorithms continue to advance, the amount of feature map data moving between layers grows and represents an increasingly large amount of the data movement associated with the whole algorithm. For example, 25% of the overall data used in the convolutional layers of AlexNet [8] (2012) were feature map data (the rest being the filter weights); in VGG [17] (2014) and GoogLeNet [18] (2014), the feature map data increased to over 50%.

To illustrate this, Figure 2 shows the size (in MB) of the feature maps (input and output) and the filter weights of each

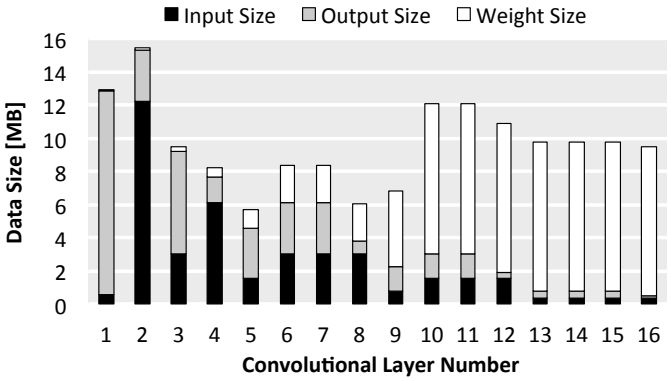


Fig. 2. Input, output, and weight sizes for different convolutional layers of VGGNet-E. This data combines each pooling layer with the prior convolution layer; for example, layer 4 encompasses one convolutional and one pooling layer.

of the convolutional layers of the VGGNet-E network [17].[1] The height of each bar represents the amount of data that must be transferred to and from DRAM (feature maps and weights) when an accelerator iteratively evaluates the layers. In the early layers, the size of the input and output feature maps dominates. For example, the first convolutional layer requires 0.6MB of input and 7KB of weights; it produces 12.3MB of output feature maps. This 12.3MB is then used as the input of the following layer (along with 144KB of weights). Performing the evaluation of the network one layer at a time requires storing the entire 12.3MB to DRAM only to immediately read back the same data (reordered) as the input of the following layer, and then repeating this back-and-forth data shuffling for every subsequent layer.

We see that, as the layers progress, the relative amount of data transfer allocated to the feature map data decreases. In the first eight layers, the sum of the inputs and outputs is much higher than the weights; beyond that, the weights dominate. (We observe a similar pattern in other common CNN structures, such as AlexNet [8].) We focus this work on reducing the amount of DRAM transferred between the early layers.

In this paper, we demonstrate layer fusion, our technique to minimize the off-chip data movement between layers by re-organizing the evaluation process. For example, our results show that, on an FPGA implementation of the first five convolutional layers of VGGNet-E (along with adjacent pooling, padding, and ReLU layers), we reduce the total data transfer required from 77MB to 3.6MB (a 95% reduction) at the cost of only 362KB of extra on-chip storage.

III. Fused-Layer CNN Accelerators

This work identifies a key opportunity in restructuring the CNN evaluation by fusing the computation of adjacent layers, largely eliminating the off-chip feature map data transfer. We develop an evaluation strategy that maximizes data reuse by forgoing the prevalent assumption that all intermediate feature maps must be stored in off-chip memory. Our design primarily targets the early convolutional layers of the networks, whose data transfers consist predominantly of the feature map data. In our approach, we fuse two or more convolutional layers into a single unit. Then, only the input feature maps of the first fused layer are transferred from DRAM. As these initial feature maps are read from memory, we compute the intermediate values of all of the fused layers that depend on the data region being read; we do not write any intermediate values out to off-chip memory. Only the output feature maps of the last fused layer are retained in their entirety. These last output feature maps are either written to off-chip memory or simply retained in an on-chip memory (if they are small enough).

A. Overview

The key to the layer fusion technique is exploiting the locality in a convolution's dataflow. Each output value computed in a convolutional layer depends only on a small window of that layer's inputs. We leverage this observation by devising an evaluation strategy where the first fused layer computes its outputs in the order that they will be needed by the second fused layer. This allows the data to be passed directly from one layer to the next, without needing to be transferred off and back on chip; once the next layer has finished consuming the intermediate data, they are discarded.

Figure 3 demonstrates the layer fusion process with an example that fuses two convolutional layers together (which we will refer to as Layer 1 and Layer 2). Note that, although the example in the following discussion focuses specifically on fusing two layers, the general form allows for more than two to be merged in an analogous way. At the top, we see the input feature maps, which are the input to Layer 1. These inputs comprise N different 7×7 feature maps. Each of the two layers convolves its feature maps with 3×3 kernels; Layer 1 has M filters of $3 \times 3 \times N$ weights, while Layer 2 has P filters of $3 \times 3 \times M$ weights.[2] In this example, all filters are applied with stride $S = 1$, although this is not a constraint of our layer fusion technique.

Layer 1 operates on a *tile* of its input feature maps, consisting of $5 \times 5 \times N$ input values (the black dashed outline labeled "tile 1" and extending "down" through all N maps). This means that $5 \times 5 \times N$ words are brought from off-chip memory and stored in on-chip buffers. Layer 1 then convolves all M of its filters (each $3 \times 3 \times N$) across this tile, producing the $3 \times 3 \times M$ region illustrated with a black dashed outline in the intermediate feature maps (and extending downward through all M feature maps). Then, Layer 2 is able to use these $3 \times 3 \times M$ values to produce $1 \times 1 \times P$ outputs (the black circle and the points extending downward) in the output feature maps.

[1] In this diagram, we assume that each subsampling (pooling) layer is merged into its preceding convolutional layer. Because subsampling is a local operation that reduces the amount of data, this always reduces bandwidth without any drawback.

[2] Because we focus on layers with relatively small filter size, we assume all filter weights are stored on chip.

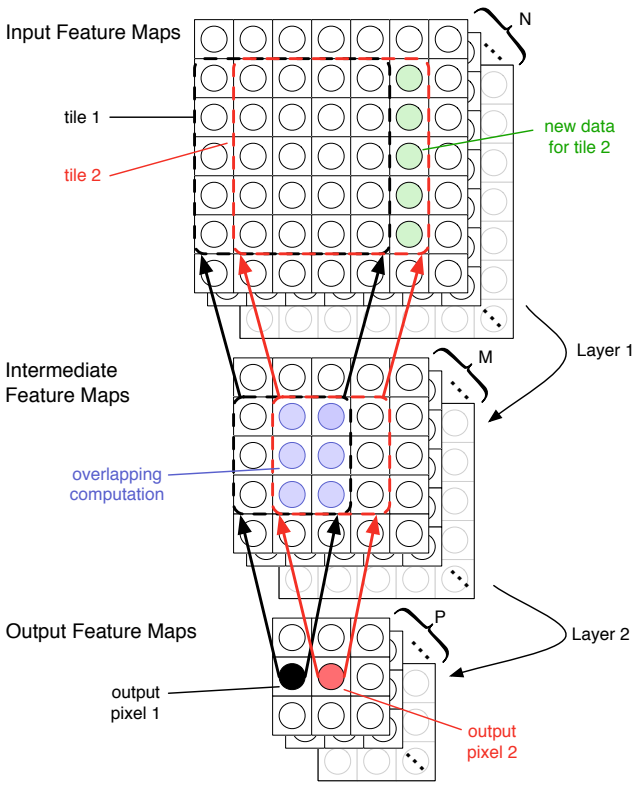


Fig. 3. Example of fusing two convolutional layers.

To reason about this process, we start from a single location in the output and trace backwards to find the region of the input feature maps that it depends on. If the layers are visualized spatially, this process creates a *computation pyramid* across multiple layers of feature maps.

Once input tile 1 (black outline) is loaded on chip, we compute the entire pyramid of intermediate values without transferring any additional feature map data to or from off-chip memory. When we reach the tip of the pyramid (the end of the fused layers), only the values in the last output feature maps are retained.

After finishing the computation of a pyramid, it is not necessary to load an entire new input tile (pyramid base) to continue. Instead, when the pyramid's base only moves by one space, it is possible to load only one new column of the input tile (while discarding the left-most column of the old tile). The green circles in the input feature maps show the new input data that must be loaded as we move the base by one location. The new input tile is indicated with a red dashed outline, forming a new red pyramid. This pyramid is evaluated in the same way, yielding output pixel 2 (the red point) in the output feature maps (extending through all P maps).

Critically, some intermediate values (the blue dots) in the intermediate feature maps are needed for computing both the black and red outputs. Because their pyramids overlap, a number of intermediate values are used to compute both sets of output values. There are two possible approaches to handle this situation. We can either *recompute* the values each time they are needed or cache and *reuse* the intermediate results while computing the next pyramid. Recomputing the values obviously adds extra arithmetic operations, but has the advantage of simplicity; each pyramid's internal dataflow is the same. Caching the intermediate results saves this extra computation, but requires on-chip buffering and makes the computation for each pyramid irregular because some pyramids must perform more computation than others.

B. Exploration Framework

We develop a technique for evaluating the costs and benefits of the fused-layer approach described in Section III-A. Based on the pyramid's dimensions, we evaluate the costs in terms of the required storage and arithmetic operations, as well as the benefit in the amount of off-chip data transfer avoided. Given a set of layers to fuse, we start from the final layer and work backwards to find the dimensions of the pyramid.

Assume a convolutional stage takes as input N feature maps and produces M output feature maps. Let the size of the pyramid at the output of a given layer be $D \times D \times M$ (by construction, if this is the final layer of a pyramid, then $D \times D = 1 \times 1$, a single value over each of the M output feature maps). We compute the pyramid size at this layer's input as $D' \times D' \times N$, where we have M convolutional filters, each of size $K \times K \times N$, applied with stride S, and $D' = SD + K - S$.

If the layer performs pooling and not convolution, we use the same equation, but set $K \times K$ to the size of the pooling window, S to its stride, and $N = M$. (Because the pooling operation is performed localized over small tiles, we always fuse the pooling layer into the previous convolutional layer, as it saves bandwidth at virtually no cost.)

Following this procedure, we can analyze the effect of fusing two or more layers by starting from the output and working backwards to calculate the dimensions of the pyramid at each level (i.e., the values at each level upon which the final outputs depend). Based on this knowledge, we can evaluate the relative costs of *recomputing* the shared data points versus adding extra buffers to locally store and *reuse* them in terms of the added computations or memory they require (respectively).

We can determine the cost of recomputation simply by examining two consecutive pyramids (e.g., the black and red pyramids in Figure 3) and examining the locations where they overlap (e.g., the $6M$ blue circles, six in each of the M feature maps). We then count the number of arithmetic operations required to compute each overlapping point, based on the dimensions of the convolution that produces it. Summing these values gives the arithmetic overhead of recomputing intermediate values for each pyramid.

Similarly, we can compute the cost of the reuse method, where a pyramid's intermediate values are stored on chip, in terms of the extra memory required. As before, we examine the same intermediate values that are shared by two consecutive pyramids. Now, rather than counting the operations that would be required to redundantly compute them as above, we instead count the amount of buffering they require. If the size of the

pyramid at the output of a given layer is $D \times D \times N$, and this layer convolves the input with a filter of size $K \times K$ with stride S, then the reuse model will require storage of $D \times (K - S) \times N$ elements on the right side of the tile (to be reused by the pyramids on the right) and $(K - S) \times D \times N$ elements at the bottom (to be reused by pyramids in the next row).

Quantifying the benefits of these methods is straightforward. For each intermediate feature map within the fused-layer pyramid, we can count the data transfer saved by avoiding writing and reading intermediate feature maps to off-chip memory. For example, by fusing layers 1 and 2 in Figure 3, we avoid writing and reading back the intermediate feature map ($3 \times 3 \times M$ points) for each CNN evaluation.

C. Recomputing vs. Storing

Based on the model above, we can evaluate the relative merit of the reuse and recompute models: is it better to store intermediate results or to recompute them? At first glance, it may appear that these methods are roughly equivalent, where the intermediate data can be recomputed or stored. However, it's important to remember that computing each of these values requires many operations, and that each intermediate point is used multiple times in the next layer.

For example, in Figure 3, the $6M$ blue values in the intermediate feature maps can be either reused or recomputed. Each of these values was computed as a convolution of the previous feature map and a $3 \times 3 \times N$ filter. Therefore, each of the $6M$ blue points required $9N$ multiplications and additions (including the layer's bias values), for a total of $6M(9 + 9)N = 108MN$ arithmetic operations. Furthermore, these $6M$ values will be used multiple times, first as the pyramid's base moves from left to right, and again as it moves down to the subsequent rows. (As a reference point, the first convolutional layer of VGGNet-E has $M = 64$ and $N = 3$; its second layer has $M = 64$ and $N = 64$.) Specifically, for a $K \times K$ convolutional kernel applied with stride S, each point (except those close to the edges) will be used in $(K/S) \times (K/S)$ pyramids. In the reuse model, once a value is computed and stored, it is reused each of these subsequent times, while the recomputation method will perform redundant recomputations for each use.

This example suggests that the recompute method may be much less efficient than the reuse strategy. To evaluate this fully, we compare the methods on real-world networks using the same procedure. Considering the two approaches for AlexNet, we find that a relatively small amount of storage can allow reuse to replace a relatively large amount of recomputation. For example, when fusing the first two layers of AlexNet, the recompute method would need to perform an extra 678 million multiplications and additions to avoid off-chip transfer, an 8.6x increase in the overall number of arithmetic operations. On the other hand, the reuse model only requires 55.86KB of additional on-chip storage to avoid the same off-chip transfer without performing additional computation.

As the network depth increases (that is, we consider fusing more layers), the difference between the two methods grows more extreme. For example, fusing all 19 convolution and pooling layers of VGGNet-E [17] would require 470 billion extra multiplications and additions in the convolutional layers, a 9.6x increase in the overall arithmetic operation count; alternatively, storing the intermediate data for reuse requires only 1.4MB of storage.

We note that in other contexts where the number of computations is relatively small, the recompute method may become useful. As one example, recurrent neural networks (used in natural language processing for language modeling) use small linear layers across multiple time steps. In examples like this, it may be preferable to recompute intermediate values rather than storing them. However, for typical CNNs targeting computer vision applications, such as the AlexNet and VGG networks we consider in this work, the costs of the recomputation model are prohibitive, while the storage costs of the reuse model are relatively small. Based on this analysis, in the remainder of this work, we focus solely on the reuse model.

D. Partitioning Networks for Layer Fusion

Although our example (Figure 3) illustrates fusing two convolutional layers, fusing more layers is analogous. As the number of fused layers increases, the benefits (reduction of data transferred to and from DRAM) increase, but so do the costs (on-chip memory required or redundant computation performed). Thus, there is a tradeoff between the costs incurred and the benefits. We can consider the case where all layers are fused into a single pyramid as an extreme: increasing costs by the largest amount to save the most bandwidth. However, we can also choose other tradeoff points, decomposing the layers using more than one pyramid.

Figure 4 illustrates two examples on a four-layer network.[3] On the left, all layers are fused into a single pyramid, resulting in minimum data transfer (only the input data for layer 1 is loaded and the final output values of layer 4 stored to DRAM). However, the input tile size at layer one and the intermediate feature maps within the pyramid will require significant storage (or recomputation). On the right, we consider decomposing the layers into two pyramids. This organization has greater off-chip memory transfer, because layer 3's output must be stored to DRAM and then read-back to compute the pyramid for layer 4. The benefit of this multi-pyramid approach is that the on-chip storage for the reuse model (or the amount of recomputation if using the recompute model) will be reduced, as the input tile and intermediate results are smaller. In this way, we obtain a space of tradeoffs: at one extreme, all layers are fused into a single pyramid. At the other extreme, where each layer is its own pyramid, the system is evaluating the CNN in the traditional layer-by-layer approach.

Given a CNN, we examine this tradeoff space by exploring the different possible ways to partition the network. We

[3]For simplicity, the illustration shows the layers as two-dimensional objects, omitting the third dimension (e.g., the M, N, and P dimensions in Figure 3).

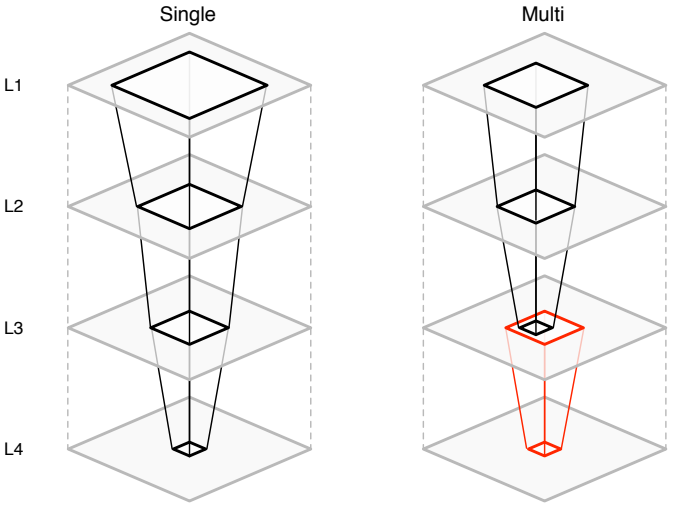


Fig. 4. Example of a single pyramid and a multi-pyramid applied over four layers (L1–L4).

consider all possible locations to start and stop pyramids and evaluate the above model for each set of possibilities. In Section V-B, we will show the results of an experiment that explores different options using the reuse method on portions of the AlexNet and VGGNet-E CNNs.

IV. Accelerator Architecture

We evaluate fused-layer CNN accelerators by implementing a hardware prototype. As explained above, we utilize the reuse method, where small amounts of intermediate data are stored (see Section III-C). We use the Vivado HLS (high-level synthesis) tool that transforms a C++ implementation into hardware, guided by `#pragma` annotations in the C++ source code. Our annotations ensure that the resulting RTL precisely matches the intended CNN evaluation strategy, orchestrating all computation and data movement cycle by cycle; we rely on the HLS tool to automatically handle pipelining of the arithmetic units and DRAM transfers.

A. Baseline CNN Accelerator

We illustrate the datapath of our baseline CNN accelerator (based on [19]) in Figure 5. The design employs loop transformations, such as loop reordering, tiling, and unrolling, to reorder computations and memory accesses, increasing throughput and reducing data transfer [19].

The pseudo-code outlining the structure of the implementation is presented in Listing 1. The in, out, and $weights$ arrays represent on-chip buffers for input, output, and weight data, respectively. These buffers function as data caches to reduce off-chip memory access; copying data in or out of these buffers (performed outside of the listing) is done using double-buffering to overlap data transfer with computation, thus requiring provisioning each memory with twice the capacity. The loops M and N are tiled with factors T_m and T_n respectively. These tiling factors control the amount and order in which data must be transferred.

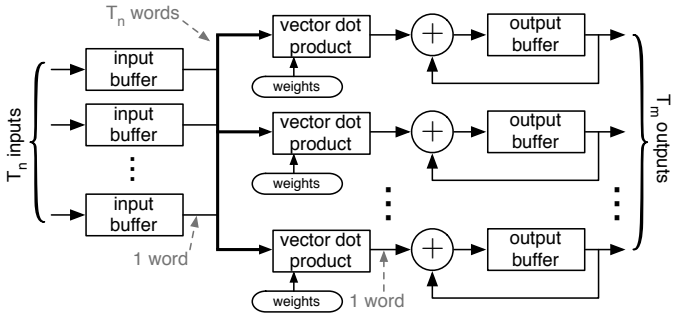


Fig. 5. A baseline CNN accelerator. Each dot-product unit takes T_n inputs and T_n weights and produces one output.

```
compute<Tm, Tn>(in[Tn][(Tr−1)∗S+K][(Tc−1) ∗ S+K] ,
                out[Tm][Tr][Tc],
                weights[Tm][Tn][K][K],
                Tr, Tc ):
 for (m = 0; m < M; m+=Tm )
 for (n = 0; n < N; n+=Tn )
  for (r = 0; r < Tr; r++)
  for (c = 0; c < Tc; c++)
  for (i = 0; i < K; i++)
  for (j = 0; j < K; j++)
  for (tm = 0; tm < Tm; tm++)  #UNROLL
   if (n == 0)  out[m + tm][r][c] = bias[m + tm]
   for (tn = 0; tn < Tn; tn++)  #UNROLL
    out[m + tm][r][c]  +=  weights[m + tm][n + tn][i][j]
                        ∗  in[n + tn][S ∗ r + i][S ∗ c + j]
   if (i == K−1  &&  j == K−1  &&  out[m+tm][r][c] < 0)
                out[m + tm][r][c] = 0  // ReLU
```

Listing 1. Convolution module pseudo-code. Computes over a tile of dimensions T_r, T_c, unrolling the inner-most two loops (T_m, T_n) to exploit hardware parallelism.

The dimensions of the inner-most two loops (T_m and T_n) are template parameters. In hardware, these loops are fully unrolled, yielding T_m vector dot-product units, each of width T_n. An accumulation adder is used after each dot-product unit, as shown in Figure 5. The overall design uses $T_m \times T_n$ multipliers and adders.

Given a hardware resource budget (e.g., a number of FPGA DSP slices available for the accelerator), one can find the optimal T_n and T_m for a given convolutional layer. In [19], a joint optimization process is proposed to create a design that can compute *all* of the convolutional layers in a given CNN. Given a resource budget, the optimization finds the (T_n, T_m) that maximizes the aggregate performance of the accelerator.

B. Fused-Layer Implementation

We construct the fused-layer accelerator around the baseline design described in Section IV-A. For contrast, we first present the baseline CNN accelerator that uses the `compute` module in Listing 2. This accelerator is triggered repeatedly, loading the input from off-chip memory and storing the output to off-chip memory for each invocation of `compute`. Double-buffering permits this design to operate at high efficiency, pipelining the loop iterations and overlapping the `compute`

```
baseline<T_m, T_n>(R, C, T_r, T_c):
  inH = S * T_r + K − S
  inW = S * T_c + K − S
  outH = T_r
  outW = T_c
  for (row = 0; row < R; row = row + T_r)
    for (col = 0; col < C; col = col + T_c)
      load (in, row, col, inH, inW)
      compute<T_m, T_n>(in, out, weights, outH, outW)
      store (out)
```

Listing 2. Pseudo-code for baseline CNN Accelerator

```
fused<T_m1, ..., T_m5, T_n1, ..., T_n5, R, C>():
  for (row = 0; row < R; row++)
    for (col = 0; col < C; col++)
      calcparams (row, col)
      load (in_1, row_t, col_t, inH_1, inW_1)
      compute<T_m1, T_n1>(in_1, out_1, weights_1, outH_1, outW_1)
      reuse (out_1, in_2, B_L1, B_T1, row, col, K_2, S_2, inH_2, inW_2)
      compute<T_m2, T_n2>(in_2, out_2, weights_2, outH_2, outW_2)
      pool1 (out_2, out_p1)
      reuse (out_p1, in_3, B_L3, B_T3, row, col, K_3, S_3, inH_3, inW_3)
      compute<T_m3, T_n3>(in_3, out_3, weights_3, outH_3, outW_3)
      reuse (out_3, in_4, B_L4, B_T4, row, col, K_4, S_4, inH_4, inW_4)
      compute<T_m4, T_n4>(in_4, out_4, weights_4, outH_4, outW_4)
      pool2 (out_4, out_p2)
      reuse (out_p2, in_5, B_L5, B_T5, row, col, K_5, S_5, inH_5, inW_5)
      compute<T_m5, T_n5>(in_5, out_5, weights_5, outH_5, outW_5)
      store (out_5)
```

Listing 3. Pseudo-code for Fused-Layer Accelerator

operation of each tile with the `load` operation of the subsequent tile.

On the other hand, the fused-layer CNN accelerator that we propose performs computation for multiple layers, eliminating all off-chip data transfer of intermediate data and writing the output feature maps only after finishing the computation for all fused layers. This is illustrated in the pseudo-code of Listing 3, which instantiates a separate `compute` module for each layer being fused. We note that the amount of computation performed by the reuse-model fused-layer accelerator and the baseline accelerator are identical; our contribution and the fundamental difference between the designs is minimizing bandwidth usage by avoiding unnecessary data transfer on and off chip between processing the layers.

The `calcparams` module runs first to determine the R and C dimensions with which the `compute` module is called for each fused layer. Between each `compute` operation, the `reuse` module is invoked to prepare the input by combining data from the preceding convolution output and the new sliver of data from the `load` operation that transfers data from off-chip memory (see Figure 3). We include pooling layers and padding layers, needed to correctly perform the CNN computation. Because pooling and padding are not computationally intensive, prior work tends to ignore these layers, concentrating on constructing accelerators for the convolutional layers. However, when fusing layers, we include these operations for completeness of the full system. Notably, while the baseline accelerator accepts M, N, K, R, C as parameters, giving it the freedom to operate on layers of any dimensions, the fused accelerator is specialized for a specific CNN and hard-codes these values to achieve its efficiency benefits.

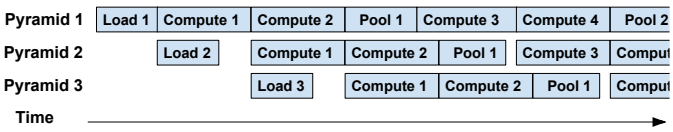


Fig. 6. Pipelining applied to fused-layer CNN accelerator

The `calcparams` module is configured at design time with the values of X, Y, S_x, and S_y (pyramid base width, height, and stride between adjacent pyramids) based on the algorithm described in Section III-B. Using these values, at each iteration, the row and col values determine the input data to transfer from off-chip memory (row_t, col_t, inW_1, inH_1) and the dimensions of each layer's computation according to the following formulas:

$$row_t = \begin{cases} Y + (row-1)S_y - (K-S), & \text{if } row > 0 \\ 0, & \text{if } row = 0 \end{cases}$$

$$col_t = \begin{cases} X + (col-1)S_x - (K-S), & \text{if } col > 0 \\ 0, & \text{if } col = 0 \end{cases}$$

$$inW_n = \begin{cases} X, & \text{if } n = 1 \text{ and } col = 0 \\ S_x + K - S, & \text{if } n = 1 \text{ and } col > 0 \\ outW_{n-1}, & \text{if } n > 1 \end{cases}$$

$$inH_n = \begin{cases} Y, & \text{if } n = 1 \text{ and } row = 0 \\ S_y + K - S, & \text{if } n = 1 \text{ and } row > 0 \\ outH_{n-1}, & \text{if } n > 1 \end{cases}$$

$$outW_n = \frac{inW_n - K}{S} + 1$$

$$outH_n = \frac{inH_n - K}{S} + 1$$

Finally, we note that the fused accelerator is pipelined to overlap the computation layers as shown in Figure 6, starting processing for pyramid two as soon as pyramid one completes its first stage. To achieve effective use of the FPGA resources allocated to the accelerator, the pipeline stages comprising the `compute` modules must be balanced. This is achieved by finding an appropriate set of T_m and T_n unroll factors for all of the layers; larger values increase the parallelism within the module, enabling the corresponding layer to finish more quickly. We limit our design space exploration to the plausible designs by considering the number of FPGA resources (DSPs), governed by

$$\sum_{i=1}^{layers} T_{mi} \cdot T_{ni} \cdot (DSP_{add} + DSP_{mul}) \leq \text{available DSPs}$$

where DSP_{add} is 2 and DSP_{mul} is 3, based on single-precision floating point units on the Xilinx Virtex-7 devices.

```
copy(src, dst, H, W, srcX, srcY, dstX, dstY)
  for (ch = o; ch < channel; ch++)
    for (row = 0; row < H; row++)
      for (col = 0; col < W; col++)
        dst[ch][dstY + row][dstX + col] =
        src[ch][srcY + row][srcX + col]

reuse(src, dst, B_L, B_T, row, col, K, S, H, W):
  if (row == 0 && col == 0)
    copy(src, dst, H, W, 0, 0, 0, 0)
  else if (row == 0)
    copy(B_L, dst, H, K−S, 0, 0, 0, 0)
    copy(src, dst, H, W−(K−S), 0, 0, 0, K−S)
  else if (col == 0)
    copy(B_T, dst, K−S, W, 0, col, 0, 0)
    copy(src, dst, H−(K−S), W, 0, 0, K−S, 0)
  else
    copy(B_L, dst, H, K−S, 0, 0, 0, 0)
    copy(B_T, dst, K−S, W−(K−S), 0, col + (K−S), 0, K−S)
    copy(src, dst, H−(K−S), W−(K−S), 0, 0, K−S, K−S)

  copy(dst, B_L, H, K−S, 0, W−(K−S), 0, 0)
  copy(dst, B_T, K−S, W, H−(K−S), 0, 0, col)
```

Listing 4. Pseudo-code for managing reuse buffers

Within the plausible designs, we perform an exhaustive search, estimating the cycle count for each layer with

$$Cycles_i = \frac{M_i}{T_m} \cdot \frac{N_i}{T_n} \cdot outW_i \cdot outH_i \cdot K^2$$

We select the option that has the minimal cycle count difference across all layers, ensuring the best pipeline balance.

C. Managing Fused-Layer Data

The key to the fused-layer strategy for CNN evaluation is in the management of intermediate data. In our implementation, this functionality is provided by the intermediate data buffers and the `reuse` module that manages them. We present the implementation of the `reuse` module in Listing 4. This module reads the results of the previous pyramid's computation from the reuse buffers and replaces them with the results of the current pyramid's computation for use by the subsequent pyramids.

The (row, col) position determines which reuse buffers are used. In the common case of operating on the "middle" pyramid (row > 0 and col > 0), results from both the B_L (buffer left) and B_T (buffer top) are used to populate the next layer's input. B_L data from position (0, 0) of height H and width $K-S$ are copied to `dst` at position (0, 0), and B_T data from $(col+K-S, 0)$ of height $K-S$ and width $W-(K-S)$ are copied to position $(K-S, 0)$. The remaining data are drawn from the `src` buffer which holds the current pyramid computations. In the less common cases ($row = 0$ or $col = 0$), data from one of the reuse buffers (B_T or B_L) are omitted. On the first iteration ($row = 0$ and $col = 0$), all data are taken from the `src` buffer and reuse buffers are not used.

V. Exploring Layer Fusion

To understand how fusing convolutional layers affects the on-chip storage and off-chip bandwidth requirements of real-world CNNs, we built a tool to evaluate the technique on CNN structures and use it to explore the resulting tradeoffs.

A. Exploration Tool

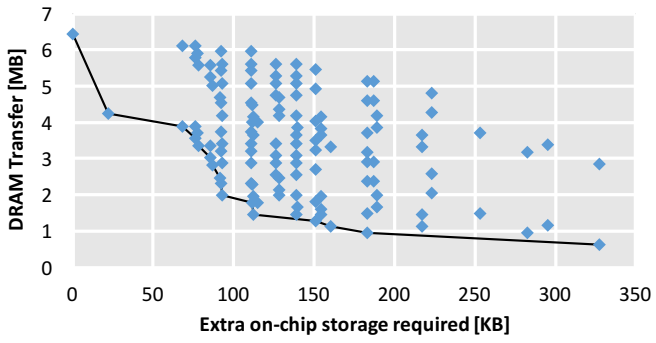
Based on the modeling technique described in Section III-B, we constructed a tool for exploring the tradeoffs of fused-layer CNN accelerator designs. To enable the tool to easily evaluate different CNN networks, we constructed it by extending the Torch machine learning framework [4], a popular system for working with deep learning algorithms. Our tool reads a Torch description of a CNN and analyzes the costs (in terms of added on-chip memory capacity or added arithmetic operations) and benefits (off-chip data accesses saved) for all possible pyramids and combinations of pyramids. The system is able to quickly enumerate and evaluate all possible design options; even for the large VGGNet-E network, the entire design space is explored in just a few minutes on a single CPU core.

B. Tradeoff Evaluation

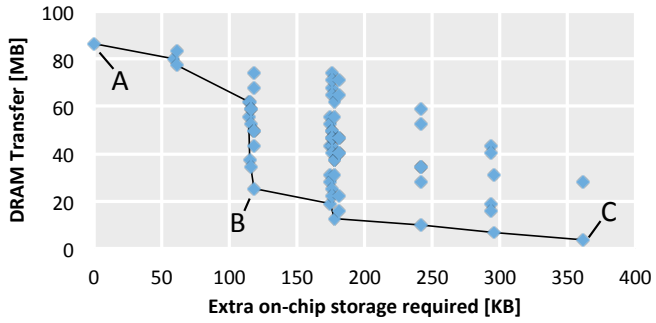
We explore the effects of applying layer fusion to real-world CNN algorithms using the evaluation tool described in Section V-A. We determine the costs (additional on-chip storage) and benefits (off-chip communication saved) for all possible groupings of fused layers.

For a given network, there are a number of ways which one may choose to fuse layers into distinct groups. Given a network with ℓ layers, there are $2^{\ell-1}$ possible ways to fuse these layers (including the extreme cases where all layers are fused into one layer, and where no layers are fused). For example, if a network has three layers, we can choose to organize the layers in groups of $(1, 1, 1), (1, 2), (2, 1)$, or (3). Although we typically expect that the best solutions involve fusing pooling layers with the convolutional layers preceding them (because pooling is an inexpensive local operation that reduces the data size), for the purposes of this analysis, we treat them as independent layers, which may or may not be merged; this allows the optimization to consider their effects.

For each network, we enumerate all possibilities and compute how much data must be transferred to and from DRAM and how much on-chip buffering is required. Figure 7 shows these results for AlexNet and VGG. The AlexNet CNN has five convolutional layers and three pooling layers; there are 128 possible combinations of different ways to fuse layers. For VGG, we consider fusing the first five convolutional layers and two pooling layers, giving 64 possible combinations. Each point on the graphs represents one possible configuration. The x-axis value indicates the on-chip storage cost of fusing the layers—the amount of extra storage required to hold the intermediate data between the fused-layers. The y-axis indicates



(a) AlexNet



(b) VGGNet-E

Fig. 7. The relationship between off-chip data transfer required and additional on-chip storage needed for fused-layer CNNs AlexNet and VGGNet-E.

each design's data transfer requirement per image.[4] The best design points are those closest to the origin, representing an ideal combination of low bandwidth and storage costs. Most of the configurations are sub-optimal—they are dominated by designs that are better on one axis and equal or better on the other. In both graphs, a solid black line connects the Pareto optimal designs.

These Pareto optimal points represent the tradeoffs that a designer may consider. For example, point A in Figure 7(b) has the lowest on-chip storage cost; it represents a layer-by-layer design that incurs no layer-fusion costs and transfers 86MB of data. Point C represents another extreme, where five convolutional layers are fused and only the input and final output feature maps are transferred. This design transfers only 3.6MB per image, a 24x reduction in DRAM traffic, but requires 362KB of on-chip memory for intermediate results. Other points between these extremes may represent attractive tradeoffs. For example, point B transfers 25MB of data, but requires only 118KB of extra on-chip storage.

VI. FPGA Evaluation

We demonstrate and benchmark our fused-layer CNN accelerators based on the HLS methodology from Section IV. We evaluate layer fusion for the early layers of AlexNet and VGGNet-E, and compare the results to those obtained using the methodology in [19].

A. Experimental Setup

We construct and evaluate two fused-layer CNN accelerators. First, we target AlexNet and compare it to the non-fused results presented in [19]. Then, we evaluate how the layer fusion technique scales to larger CNNs by accelerating VGGNet-E and comparing the results to a baseline design we produced following [19]'s methodology.

We use Xilinx Vivado HLS 2015.4.2 to generate designs using high-level synthesis targeting a Xilinx Virtex-7 XC7V690T FFG1761-3 FPGA. For ease of comparison with prior work, we use single-precision floating point for all designs, and we size our design to match the resources and target clock frequency of [19] as closely as possible. For each design, we report the HLS tool's FPGA resource consumption estimates, the cycle counts, and the amount of data that must be transferred to/from DRAM for each image.

Lastly, we remind the reader that in this paper we are specifically focusing on the convolutional and pooling layers of CNNs where most of the data to be transferred are input and output feature maps. These layers, which occur early in typical CNNs, have relatively small amounts of filter weights; because of this, the weights easily fit into on-chip storage in their entirety for these layers.

B. Results and Comparison

We evaluate our fused-layer accelerator strategy on the AlexNet and VGGNet-E networks.

AlexNet. Based on the analysis in Section V-B, we fuse AlexNet's first two convolutional layers; these are the layers with the largest feature maps, which require most of the feature map bandwidth. Although we are evaluating our design on a larger Virtex-7 FPGA than [19], we constrain our exploration tool to use designs with roughly the same memory and arithmetic resources. Additionally, we include AlexNet's nonlinear layers, which are critical to real-world CNN algorithms: a rectified linear operation called ReLU (which performs the function $f(x) = \max(x, 0)$ on each output value), a zero-padding around the inputs, and a 3×3 pooling layer (with stride 2) that keeps the largest value in each 3×3 region of the output. In all, we fuse two convolutional layers, two ReLU layers, two padding layers, and one pooling layer.

In order to directly compare with [19] we also omit AlexNet's normalization layer. This has a very small effect; normalization represents a tiny portion of the CNN computation and requires minimal extra FPGA resource utilization. Our fused-layer accelerators can trivially integrate normalization into the design; it would appear as a single additional stage in the Figure 6 timing diagram, without affecting the overall accelerator throughput.

Table I gives a comparison between our fused-layer accelerator and an accelerator derived from [19]. When demonstrating the differences between these, we strive to avoid any

[4]Data transfer values can be converted to bandwidth by multiplying by the target throughput. For example, if an accelerator targets 50 images/second, and the graph shows an off-chip transfer of 100MB, this would require 5 GB/sec. bandwidth.

TABLE I
COMPARISON OF OUR FUSED-LAYER ACCELERATOR FOR THE FIRST TWO CONVOLUTIONAL LAYERS OF ALEXNET WITH A BASELINE DESIGN DERIVED FROM [19].

	Fused-Layer	Baseline
KB transferred/input	688	962
Cycles$\times 10^3$	422	621
BRAMs	1,124	1,046
DSP48E1	2,401	2,240
LUTs	273,367	186,251
FFs	306,990	205,704

TABLE II
COMPARISON OF OUR FUSED-LAYER ACCELERATOR FOR THE FIRST FIVE CONVOLUTIONAL LAYERS OF VGGNET-E WITH A BASELINE DESIGN DERIVED FROM [19].

	Fused-Layer	Baseline
MB transferred/input	3.64	77.14
Cycles$\times 10^3$	11,665	10,951
BRAMs	2,509	2,085
DSP48E1	2,987	2,880

unfair advantages for the fused-layer approach. Therefore, we make several adjustments or improvements to the baseline. First, [19] provides the design parameters that jointly optimize over *all five* AlexNet convolutional layers. Because we only consider the first two convolutional layers in the fused-layer accelerator, we repeat [19]'s optimization for just the same two layers. Second, the results given in [19] do not include any of the non-linear layers (pooling, ReLU, or padding). Although these layers contribute very little computation, we note that the pooling layers greatly *reduce* the amount of data that must be transferred off chip. Thus it would be unfair to compare the bandwidth of our fused-layer design (which includes pooling) to theirs. Therefore when we calculate the data transfer requirements of [19] we include pooling, and account for its cost as only 22 additional BRAMs to implement this layer. Third, we conservatively assume that the time taken to perform the nonlinear operations in [19] can be entirely overlapped with existing processing, without negatively affecting the pipeline depth or design frequency. Lastly, we directly compare against the numbers of DSP48E1 slices, LUTs, and flip-flops reported in [19] without accounting for any additional overhead caused by the added nonlinear operations. This ensures that the baseline design we compare with has all of the benefits of the nonlinear layers (reduced data transfer) without negatively affecting its performance or FPGA resource consumption.

The values for the fused-layer design in Table I are taken directly from the HLS implementation based on the methodology described in Section IV. Comparing the two designs, we see that the fused-layer method gives a 28% savings in off-chip data transfer, even when applied only to two layers. The fused-layer design exhibits small increases in the number of DSP slices and BRAMs, and an approximately 50% increase in the FPGA's LUTs and FFs used. These additional costs are due to [19] not including the non-linear layers (padding, pooling, and ReLU), and an increase in control logic complexity caused by layer fusion.

VGGNet-E. Table II shows the results of a similar comparison, but for the larger and newer VGGNet-E network [17]. To demonstrate how the performance and cost of the fused-layer strategy scale as the number of layers increases, we fuse the first five layers of VGGNet-E. In addition to the five convolutional layers, this includes two pooling layers, five padding layers, and five ReLU layers. This design corresponds to the point labeled C in Figure 7(b). As before, we construct our design using Vivado HLS and report the cycle count, off-chip transfer, and resource usage. For the baseline design, we again use the optimization approach from [19], jointly optimizing for the first five convolutional layers. We use the same conservative assumptions for the costs of the non-linear layers as for AlexNet.[5]

The off-chip data transfer for the first five convolutional layers of VGGNet-E is much higher than for AlexNet. Without the fused-layer strategy, the baseline design transfers 77MB of data between the FPGA and DRAM for every image. The fused-layer accelerator drastically reduces this memory transfer down to 3.6MB, a 95% decrease. This reduction in off-chip transfer comes at a cost of an extra 424 BRAMs (an increase of 20%), and a minor increase in DSP slices (due to the additional control logic). We use a conservative calculation for the baseline's cycle count, which does not account for the overhead of the first padding layer or the time needed to initially fill the pipeline at the beginning of each iteration. Compared with this idealized model, our fused-layer design is marginally slower, requiring 6.5% more clock cycles.

C. Discussion

Layer fusion can save bandwidth on all architectures. In this work, we demonstrated the layer fusion strategy with an FPGA prototype because it allowed us to precisely orchestrate the cycle-by-cycle data movement and exactly quantify the layer fusion benefits and costs. However, the layer fusion technique is generally applicable to all programmable systems that evaluate CNNs and to ASIC implementations that target a specific network. For example, our experiments with a C++ implementation of layer fusion for the first two layers of AlexNet achieves more than 2x speedup as compared to the layer-by-layer approach running on a desktop CPU. We similarly expect GPUs to benefit from layer fusion; however, current GPU programming abstractions make it challenging to precisely orchestrate the thread behavior and buffer management of layer fusion.

VII. RELATED WORK

Recent works on CNN performance have identified data transfer as a primary concern to achieve efficient processing

[5]We do not report the LUT or FF usage for VGGNet-E because these values are not given in [19], nor can they be estimated based on its models.

and have developed design methodologies for CNN accelerators that minimize off-chip memory accesses [10], [19].

A number of works have targeted reducing the bandwidth requirements of transferring data into on-chip buffers [10], [12] and constructing models that maximize computation while minimizing bandwidth requirements [19]. These works target minimizing the data transfer incurred for each layer individually. In [11], a mechanism for fusing a pooling layer with a preceding convolutional layer is presented. We similarly fuse the pooling and padding layers, however our main bandwidth advantage comes from fusing multiple convolutional layers. Notably, all prior designs construct a single accelerator with flexibility to execute any convolutional layer, whereas our fused-layer strategy gains efficiency by specializing the processing for different layers.

In [6], [5], [16], systolic implementations were proposed. [16] also applies parallelism within feature maps to maximize resource utilization. Unlike our work, these mechanisms don't need to handle a large amount of intermediate data that must be stored in off-chip memory.

Several ASIC implementations of CNN accelerators have also been proposed [2], [9]. These designs optimize different modules of neural networks (pooling, convolution, etc.) and other machine learning algorithms so as to minimize external data access, and they use loop tiling techniques to increase data locality. Although the primary goal of our work is to eliminate the transfer of intermediate data, we also apply similar loop tiling techniques to reduce the data transferred to/from off-chip memory for each pyramid's initial and final layers.

In [14], the authors proposed a generalized convolution engine that can be used for various computer vision and computational photography algorithms which have convolution-like data-flow patterns. Similar to our design, this work fuses multiple arithmetic operations within the convolution engine to reduce memory storage requirements. However, this work targets algorithms which perform 1D or 2D convolution, rather than the 3D convolutions needed for CNNs.

All the approaches mentioned above optimize the convolution layers, but none of them focus on reducing the off-chip memory accesses of hidden layer data. We have shown in our approach that by fusing different layers, we can significantly reduce the number of off-chip memory access.

VIII. Conclusions

Deep convolutional neural networks (CNNs) are rapidly rising in popularity across a wide array of fields, leading to significant interest in the design of accelerators for them. The conventional approach to designing CNN accelerators has been to iteratively process layers to completion, forcing off-chip storage of the intermediate data. We observed that it is possible to *fuse* the processing of adjacent CNN layers by bringing data on chip using a pyramid-shaped multi-layer sliding window. The key advantage of the fused-layer evaluation strategy rests in the ability to cache inter-layer intermediate data on chip, thereby minimizing off-chip transfer. In this work, we detailed the methodology to develop fused-layer accelerators and demonstrated the effectiveness of our approach by implementing a fused-layer CNN accelerator on a Xilinx Virtex-7 FPGA. On the the first five convolutional layers of the VGGNet-E network, we showed that the fused-layer accelerator design can reduce data transfer by 95% compared to the previous state of the art.

Acknowledgment

This material is based upon work supported by the National Science Foundation (NSF) under Grants No. 1453460, No. 1452904, and No. 1533739. The experiments were conducted with equipment purchased through NSF CISE Research Infrastructure Grant No. 1405641.

References

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

[2] 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 *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014.

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

[4] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A Matlab-like environment for machine learning," in *Proceedings of BigLearn, NIPS Workshop*, 2011.

[5] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Proceedings of the Embedded Computer Vision Workshop*, 2011.

[6] C. Farabet, C. Poulet, Y. Han, J., and Y. LeCun, "CNP: An FPGA-based processor for convolutional networks," in *Proceedings of the 2009 International Conference on Field Programmable Logic and Applications*. IEEE, 2009.

[7] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "DjiNN and tonic: DNN as a service and its implications for future warehouse scale computers," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proceedings of the Advances in Neural Information Processing Systems 25 (NIPS)*. Curran Associates, Inc., 2012.

[9] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "PuDianNao: A polyvalent machine learning accelerator," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2015.

[10] M. Peemen, A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Proceedings of the 31st International Conference on Computer Design (ICCD)*. IEEE, 2013.

[11] M. Peemen, B. Mesman, and H. Corporaal, "Efficiency optimization of trainable feature extractors for a consumer platform," in *Proceedings of the 13th International Conference on Advanced Concepts for Intelligent Vision Systems*. Springer-Verlag, 2011.

[12] ——, "Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015.

[13] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. IEEE, 2014.

[14] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2013.

[15] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision*, no. 3, 2015.

[16] 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 *Proceedings of the 20th IEEE International Conference on Applications-specific Systems, Architecture and Processors (ASAP)*. IEEE, 2009.

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proceedings of the International Conference on Learning Representations*, 2014.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[19] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2015.