The Thesis committee for Daniel Levi Rauch certifies that this is the approved version of the following thesis:

Scale-CNN: A Tool for Generating Scalable High-Throughput CNN Inference Accelerators on FPGAs

SUPERVISING COMMITTEE:

Lizy K. John, Supervisor

Andreas Gerstlauer

Scale-CNN: A Tool for Generating Scalable High-Throughput CNN Inference Accelerators on FPGAs

by

Daniel Levi Rauch

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2021

Scale-CNN: A Tool for Generating Scalable High-Throughput CNN Inference Accelerators on FPGAs

Daniel Levi Rauch, M.S.E.
The University of Texas at Austin, 2021

Supervisor: Lizy K. John

In the past decade, research has shown that CNN inference can be considerably sped up via dedicated hardware accelerators. However, most existing accelerators have limited performance by only working on a single inference at a time and/or relying on slow off-chip memory accesses for hidden layers. These limitations stem from the high memory requirements of CNN inference, which can be 10s of Mb even for small networks with reduction techniques. Despite this, as Moore's law has continued to scale, this level of on-chip memory is now attainable. We propose Scale-CNN, a tool for generating multiple Pareto optimal design points for high-throughput CNN inference accelerators on Xilinx Ultrascale+ FPGAs. The Scale-CNN architecture dedicates separate hardware resources for each layer and stores all feature maps and weights on-chip, enabling a high-throughput network pipeline where each layer works on a different inference simultaneously with no off-chip memory accesses. Using Scale-CNN, we generate several accelerator IPs for Tiny Darknet on the smallest Virtex Ultrascale+ FPGA (XCVU3P) that range from 1.7 to 56.7 inferences per second utilizing 22% to 66% of FPGA resources.

Table of Contents

List of Tables

List of Figures

1 Introduction

The ever-growing demand for high-performance ML systems has spurned an explosion of industry projects and academic research on Deep Neural Network (DNN) acceleration in the past decade. As these demands have now pushed even modern-day CPUs and GPUs beyond their limits, the research has turned to focus on dedicated hardware accelerators for ML-related tasks. Convolutional neural network (CNN) inference in particular has received especially great attention from industry and academia given its widespread use in image recognition and classification applications.

Many of the most well-known works in this area are ASIC-based. While ASICs typically always dominate in performance per watt compared to alternatives, they suffer from lack of reprogrammability and are cost-prohibitive to many smaller industry and research firms. Conversely, GPUs are easily reprogrammed but lack in performance as they are general purpose and not hard-wired for DNN acceleration. FPGAs are a promising solution that bridge this gap between GPU-based and ASIC-based designs. FPGAs are both reprogrammable and can be configured into efficient dedicated accelerators for specific ML tasks.

Other academic works such as [17], [18], [1], and [7] focus on CNN inference acceleration on FPGAs. These works generally suffer from one of two notable drawbacks. First, they have a relatively low inference throughput. This is because the same hardware resources are used to execute each layer, limiting the accelerator to working on a single inference at a time. This limitation is not due to a shortcoming in these accelerators' designs but rather stems from the fact that the FPGAs targeted by these works are simply not large enough to dedicate independent on-chip resources for each layer. However, as Moore's law has continued

to scale, the latest high-end FPGAs are now large enough that such an accelerator is now possible.

Second, while the designs may be parameterizable for different networks with different dimensions, they only provide a single design point for a given network that cannot be tuned to adjust the cost/performance tradeoff. Alternatively, they may be tunable through user-settable parameters, but do not provide a solution for design space exploration of these parameters. A SoC designer for a real-time image recognition application is likely to want both high-throughput inference on the order of multiple inferences per second and a series of recommended design points for the accelerator that span a wide range on the cost/performance tradeoff curve. This would allow the designer to choose the cheapest implementation that meets a minimum system-level requirement for inference throughput.

To this end, we propose Scale-CNN, a tool that generates multiple implementations for a high-throughput CNN inference accelerator for large, high-end FPGAs. All implementations share the same high-level architecture, but vary in implementation details that yield different design points with a diverse range of cost and performance. The tool generates these points by quickly finding the Pareto optimal solutions among its design space. Scale-CNN assumes a constant influx of input images to process, making it ideal for real-time image recognition applications involving live video. We demonstrate that Scale-CNN is capable of generating implementations for Tiny Darknet [10] on the smallest Xilinx Virtex Ultrascale+ FPGA ranging from 1.7 inferences/second to 56.7 inferences/second and utilizing 22% to 66% of the FPGA's on-chip resources. These extremes correspond to 1.65 and 55.6 GFLOPS, respectively.

2 Background

Convolutional neural networks (CNNs) are a popular form of neural network widely used for image recognition and classification. The network takes in an image and outputs probabilities that indicate how strongly the network "believes" the image pertains to different pre-determined categories. They consist of multiple layers of different types connected in simple feed-forward fashion. Figure 1 shows the high-level diagram of a CNN. The most important layer in the CNN is the convolution (or CONV) layer.

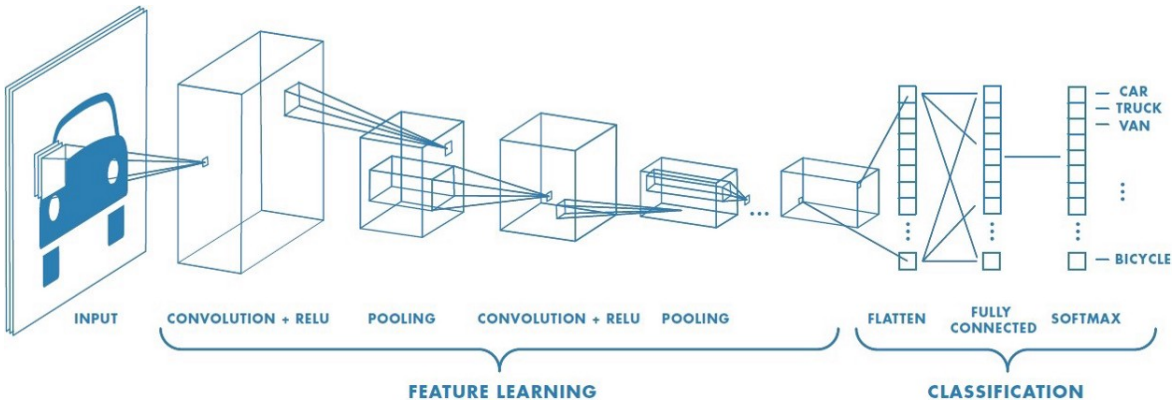

Figure 1: High-level architecture of a CNN, taken from [11]

2.1 Convolution Layers

A convolution layer consists of 3-dimension input and output feature maps (or *fmaps*), 4-dimensional weights representing a set of filters, and 1-dimensional per-filter biases. Additionally, the CONV layer may include 1-dimensional per-filter batch normalization constants. All data other than feature maps are previously learned off-line during training.

The 3D feature maps represent the image that is being fed through the network. The first two dimensions represent physical space and correspond to pixel coordinates. The third

dimension represents channels, which quantitatively reflect some aspect of the image at each point. For the first convolution layer, the channels typically represent the RGB values of the original color image. Beyond the first layer, each channel in the input fmaps represents how strongly each point in the image "reacted" to a filter in the previous layer.

In this paper, we use the notations shown in Table 1 for representing layer dimensions.

Parameter	**Description**
IH/IW/OH/OW	Input/Output Width/Height
N	# Input Channels
M	# Filters / # Output Channels
K	Filter width/height
S	Stride size

Table 1: CNN dimension notation

The 4-dimensional weights can be thought of as M separate 3-dimensional $K \times K \times N$ filters, each corresponding to a different output channel. To compute the output feature maps, each filter is "slid" across every possible position of the input feature maps. At every position, the individual weights in the filter are multiplied against their corresponding ifmap element at the same X/Y coordinate and input channel. These products are then accumulated into a sum. This sum then goes through an optional batch normalization step where it is normalized to per-filter mean and variance. Next, a per-filter bias is applied, and finally, the value is passed through an activation function such as ReLU. The formula for the convolution layer is given below (batch normalization excluded):

$$\mathbf{O}[x][y][m] = f\left(\left(\sum_{i=0}^{K-1}\sum_{j=0}^{K-1}\sum_{n=0}^{N-1}\mathbf{I}[Sx+i][Sy+j][n] \times \mathbf{W}[m][i][j][n]\right) - \mathbf{b}[m]\right) \quad (1)$$

$$0 \leq x < OW, 0 \leq y < OH, 0 \leq m < M \quad (2)$$

O, **I**, **W**, and **b** represent the output feature maps, input feature maps, weights, and biases, respectively, while f represents the activation function. The parallel nature of this algorithm makes it amenable to hardware acceleration. Each output element is computed independently with many opportunities for data reuse.

2.2 Pooling Layers

Pooling layers are used to reduce the width and height dimensions of the image. With a pooling factor of P, each $P \times P$ square of the inputs is reduced to a single output for every channel. There are two types of pooling layers commonly used in CNNs - average pooling (avgpool) and maximum pooling (maxpool), which calculate the average and maximum of all $P \times P$ elements, respectively. Scale-CNN only supports maxpool layers.

2.3 Other Layers

Other layers commonly found in CNNs include fully-connected (FC) layers and softmax layers. These layers are frequently found at the end of the network, and involve relatively little computation compared to the rest. For this reason, Scale-CNN does not implement these layers and leaves them as the CPU's responsibility.

3 Related Work

The massively parallel nature of the convolution layer means a naively-designed accelerator would need a relatively large amount of memory bandwidth for a relatively small amount of compute bandwidth. For this reason, the earliest works on CNN inference accelerators, such as Eyeriss [4] and DianNao [3], are primarily focused on minimizing off-chip memory bandwidth requirements. DianNao accomplishes this by optimizing the locality of the algorithm through a tiling approach similar to a blocked matrix multiplication. Eyeriss utilizes a systolic array [9] to maximize data reuse within the compute unit itself.

Other works similar to these two are FPGA based. In [17], the authors propose a design-space exploration technique for a systolic array-based accelerator on FPGAs similar to Eyeriss. In contrast, [18] is more DianNao-like in focusing on optimal tiling strategies.

Central to the design philosophy of all of these works is the presumption that hidden layer feature maps must either be stored off chip or must share the same on-chip memory at different times, limiting the accelerator to working on a single layer of a single inference at a time. This is a reasonable presumption to make given the sheer amount of memory required for an entire network – the sum of all feature map and weight sizes for all layers is on the order of 100s of Mb even for small CNNs. However, there are well-known strategies that can reduce these requirements considerably by more than an order of magnitude. The most common strategy is to use low-precision data types, which have been found to cause minimal degradation in inference accuracy even for drastic reductions in precision [13]. Some works take this concept to the extreme with binarized weights and activations [5]. FINN [14] proposes an accelerator very similar to Scale-CNN that uses binarized neural networks. The 1-bit operations in FINN allow for a massively parallel accelerator that can achieve

inference throughputs up to hundreds of thousands of inferences per second. However, this requires the input images to be binarized as well. Scale-CNN opts to use 16-bit half-precision floating point (FP16) for all weights and activations. This is a popular choice among other CNN inference accelerators and instantly provides a 50% reduction in memory requirements compared to traditional 32-bit single precision floats.

The second strategy used by Scale-CNN is layer fusing. A CNN accelerator that takes advantage of layer fusing is proposed in [1]. Scale-CNN always fuses maxpool layers with their previous convolution layers, and supports fusing adjacent convolution layers in very limited cases. While [1] supports fusing any number of arbitrary convolution layers together, Scale-CNN can only fuse two layers together when the second layer has a kernel size of 1x1. This requirement simplifies the data dependencies between the two layers and is discussed further in Section 8.6. Luckily, this limitation works well for Tiny Darknet which has multiple layers with 1x1 filters.

The most similar work to Scale-CNN known to the author is [8]. The accelerator proposed in this work uses a large FPGA, stores all hidden-layer feature maps in on-chip ping-pong buffers creating a network pipeline, and allows multiple network implementations to be made by parameterizing the unroll factors of different layers. However, this work does not appear to support design space exploration of these parameters on behalf of the user. Rather, the parameters are exposed as user-settable, but determining optimal values for them is left as the user's responsibility entirely. Scale-CNN goes one step further and automatically explores the unroll factors of loops within a layer to create multiple layer implementations. These layer implementations are then mix-and-matched to make multiple Pareto optimal network implementations.

Lastly, many works have proposed design space exploration techniques for determining optimal pragmas for a given HLS code. In [6], the design space is represented as a multi-dimensional lattice which is explored under the premise that Pareto optimal solutions tend to be close together. A similar approach is proposed in [20], which focuses on designs that primarily consist of multiple nested loops. This work also involves pruning the design space to reduce long exploration times. Other works, such as [19], do not explicitly propose a searching algorithm but rather provide an analysis tool for quickly estimating the cost and performance for a combination of pragmas applied to an HLS code. All of these works make two prior assumptions: 1) that the design space is so large that searching it in a reasonable time frame requires pruning and/or quick performance/cost estimation models, and 2) that the C/C++ code for different design points must be the same. Neither of these hold true for Scale-CNN. The number of options considered for each layer is relatively small (rarely larger than 30) and each option only takes a few minutes to synthesize. Additionally, allowing the code to vary between implementations provides an extra degree of flexibility for finding more optimal design points. Of course, code variations are application-specific and must be made by the designer rather than an automated process.

4 Overview

4.1 Use Model

Figure 2 details the Scale-CNN design flow. Scale-CNN is a tool that takes in a description of a network (most importantly, the dimensions of each layer) and a description of the target FPGA specifying the quantities of each resource (FFs, LUTs, etc.). It then generates a number of implementation options for the network that will not exceed the available resources on the FPGA. These options will span from cheap, slow implementations that use minimal resources to much faster ones that use a large percentage of the available resources on the chip. Using the algorithm described in Section 10.1, the tool is able to quickly find the Pareto optimal points in its design space with respect to resource cost and throughput. If the number of possible implementations is large, the tool will select a subset of design points as recommendations (see Section 11.3). The user can then select a particular design point and direct the tool to generate it and run it through the HLS toolchain. The HLS toolchain can then call the underlying FPGA toolchain to synthesize the RTL for the target FPGA. Both HLS and FPGA synthesis tools generate reports that the user can examine to verify the resource usage, latency, and clock period meet expectations.

4.2 Architecture

Figure 3 shows an overview of the Scale-CNN accelerator architecture. All design points generated by Scale-CNN adhere to this architecture. The architecture's defining feature is a hierarchical three-level pipelining structure.

At the highest level is the network pipeline, where each stage is a different layer. Each layer works on a different inference concurrently. Between each layer are large ping-pong

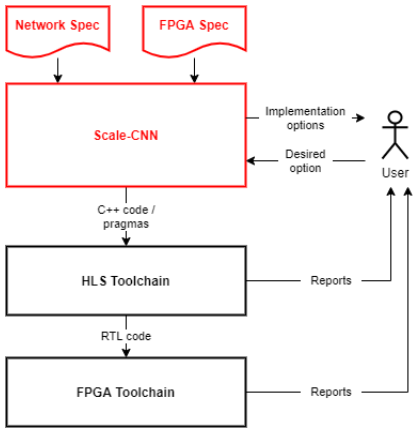

Figure 2: Use model of Scale-CNN

buffers that hold the feature maps. The beginning and end of the pipeline are pseudo-layers that read/write feature maps to/from an AXI4-Lite bus.

The next level is the layer pipeline, where each stage represents one step in the process to compute one or more output elements. This pipeline is wrapped inside a loop that iterates over all output feature maps of the layer, which then become the input feature maps for the next layer. Between each stage are smaller ping-pong buffers or FIFOs that hold the intermediate results of each step. This level also shows the weight and bias buffers. These buffers are assumed to be loaded before inference begins and remain unchanged during operation. For this reason, they do not need to be double-buffered.

The lowest level is the function pipeline. This is a "traditional" pipeline where each stage is a single clock cycle separated by FFs. Although Figure 3 only shows the pipeline for the multiplication stage, all stages in the layer pipeline are implemented as low-level function pipelines that iterate over a number of input elements.

The extensive use of pipelining at multiple levels of the design creates a flexible

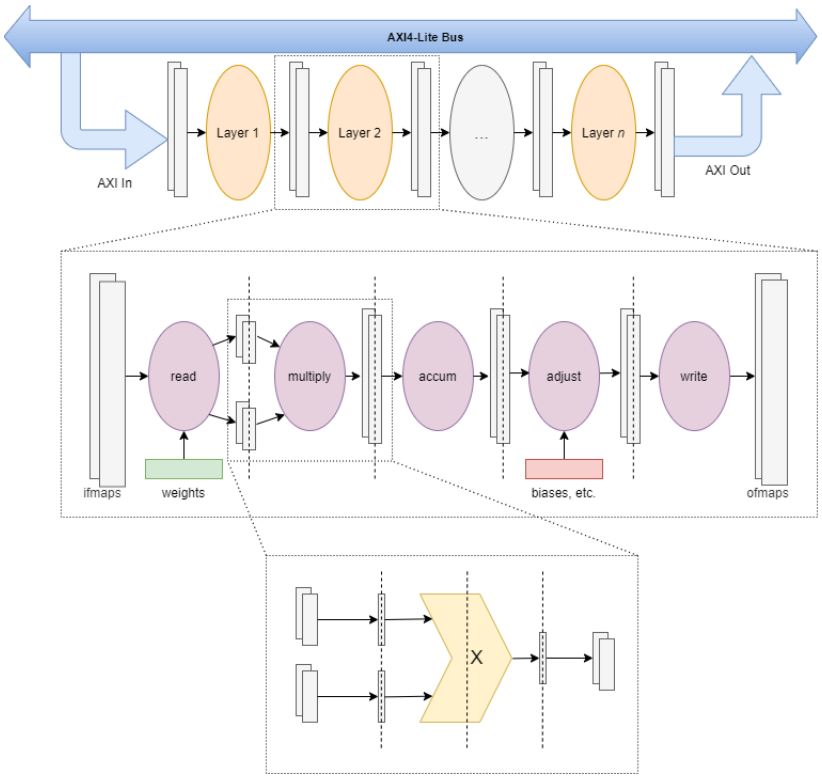

Figure 3: Scale-CNN architecture with three levels of pipelining: the network pipeline (top), layer pipeline (middle), and function pipeline (bottom)

architecture that can be easily scaled to improve performance for a higher cost while always making very efficient use of resources. High resource efficiency is achieved by ensuring the low-level function pipelines have an initiation interval of 1 and by making sure the network and layer pipelines have well-balanced stage latencies.

Scale-CNN is designed for inference only. It is assumed that the weights and biases are learned offline. Additionally, Scale-CNN is only focused on the design of the accelerator itself. It does not provide a solution for integrating the accelerator into a higher-level CPU-FPGA coprocessing system.

The rest of this paper is structured as follows. Section 5 discusses the key design decisions of Scale-CNN and their motivations in further detail, Section 6 discusses the HLS pragmas and directives used in the designs, and Section 7 discusses the cost function used in the work. Sections 8 and 9 discuss the design of individual layers and compare the synthesis results different implementations, while Sections 10 and 11 do the same for networks. Finally, we discuss future work for extending Scale-CNN and conclude the paper.

5 Key Design Decisions

While some details vary between different implementations generated by Scale-CNN, most are the same. In this section, we lay out the key design decisions that are common to all Scale-CNN-generated designs.

5.1 High Level Synthesis (HLS)

Scale-CNN designs are entirely HLS-based with code written in C++. The motivation for this is that HLS code describes the design at a higher level of abstraction than HDLs. This enables the HLS tools (and the designer) to have more flexibility in deciding the specific implementation details of the generated RTL separately from the code. This is highly beneficial since the goal is to generate different implementations for the same higher-level design behavior. Using HLS enables the C++ code for different implementations to remain largely (but not entirely) identical. Different design points primarily differ not in the code but rather in the pragmas and directives that are used to specify the implementation (see Section 6). Scale-CNN specifically uses Vitis HLS, a HLS toolchain developed by Xilinx for use with Xilinx FPGAs.

5.2 Tiny Darknet

Scale-CNN is designed to work for CNNs with layers of arbitrary dimensions. However, for this work, we focus on Tiny Darknet for a number of reasons. First, Tiny Darknet is smaller than most CNNs, lending itself well to a design that is very memory-intensive by nature. Second, Tiny Darknet layers exhibit a convenient uniformity in their dimensions. For instance, all convolution layers have filter sizes of either 1 or 3, and the number of chan-

nels for all layers except the first and last is a power of 2. These regularities are beneficial since an optimization applied to one layer can likely apply to others as well. Lastly, all layers except the last three are either convolution or maxpool layers. These last three layers are much less computationally intensive than the rest and are thus not considered for this work. Implementing these layers would best be done on the host CPU rather than wasting more FPGA resources.

5.3 Half-precision Floating Point

As discussed in section 4, all floating point data in Scale-CNN designs is represented in 16-bit half-precision floating point format with 5 exponent bits and 10 mantissa bits (FP16). The benefits of using half-precision floating point compared to single-precision is massive, yielding a 50% reduction in on-chip memory requirements and considerable reduction in FFs, LUTs, and possibly DSPs as well (depending on the implementation).

5.4 Layer Pipeline

The key aspect of the Scale-CNN architecture that distinguishes it from most existing works is the high-level layer pipeline. Whereas most CNN accelerators proposed in other works map every layer's execution to the same hardware resources (such as a systolic array), Scale-CNN dedicates separate hardware resources for each layer and double-buffers the feature maps between them. This allows each layer to work on a different simultaneously in a pipelined fashion, which is key to achieving high-throughput inference on the order of tens of inferences per second.

5.5 Off-chip Memory Accesses

In conjunction with the network pipeline, the other key feature that enables such high-throughput inference is that there are no off-chip memory accesses for hidden layers. All feature maps, weights, and biases are stored in on-chip RAMs. Other works rely on slow off-chip accesses for intermediate results, which risk bottlenecking the accelerator and under-utilizing compute resources. This is a classic example of the memory wall problem [15]. These other works are largely focused on mitigating the issue by minimizing these memory accesses as much as possible. Scale-CNN avoids the problem entirely by using very large high-capacity FPGAs which eliminate the need for off-chip accesses altogether.

5.6 Memory Requirements

Until recently, a CNN inference accelerator with no off-chip memory accesses was not feasible due to the sheer amount of on-chip memory required to hold all feature maps and weights, even for small networks. The minimum memory requirements for implementing Tiny Darknet with the Scale-CNN architecture can be found in Table 2. This shows that the network requires 46 Mb of on-chip storage for feature maps and weights. With the exception of the absolute largest Virtex 7-Series devices, Xilinx FPGAs did not have this level of on-chip memory capacity until the introduction of the Ultrascale and Ultrascale+ families in the mid-2010s. With on-chip memory capacities now in the hundreds of Mb, the Scale-CNN architecture is feasible not only for Tiny Darknet but larger networks as well.

Scale-CNN is exclusively targeted for Ultrascale+ FPGAs. These were chosen because they have a new type of on-chip RAM called "UltraRAMs", or "URAMs". UltraRAMs are higher capacity than traditional Block RAMs, with 288 Kb per instance compared to 18

Kb. Feature map arrays that are several Mb large would require hundreds of BRAMs to implement, which could cause significant routing congestion between the RAMs and the rest of the accelerator logic. Implementing these arrays with UltraRAMs mitigates this problem considerably. Scale-CNN places all feature map arrays in URAMs and most weight arrays in BRAMs. However, especially large weight arrays may be placed in URAMs if appropriate. This is discussed further in Section 10.2.

6 HLS Pragmas & Directives

As discussed in Section 5.1, HLS tools allow the designer to specify a high-level behavioral description of the design in C/C++ without specifying key implementation details about the generated RTL. These details can be specified through the use of pragmas and directives. Pragmas and directives are functionally equivalent to each other but differ in form. Pragmas appear as preprocessor directives in the C/C++ code, whereas directives are more programmatic and are specified through TCL commands. Scale-CNN utilizes both forms, typically opting to use pragmas when the same optimization is applied to all implementations of a layer or network, and directives when the optimization varies between implementations. Detailed information regarding all pragmas and directives supported by Vitis HLS can be found in [16]. The main ones used by Scale-CNN are summarized here.

6.1 Loop Unrolling

By default, a for-loop in HLS code will synthesize into RTL with a single instance of the loop body. Each iteration of the loop must move through this loop body instance one at a time. Loop unrolling duplicates the logic of the loop body so that multiple iterations of the loop can execute in parallel. Loop unrolling effectively transforms the loop into a new loop with a lower trip count (number of iterations) but more work per iteration. However, assuming there are no loop-carried dependencies between iterations, the additional work per iteration is performed in parallel, causing the total loop latency to be inversely proportional to the unroll factor. Some loops can be completely unrolled, meaning the unroll factor is equal to the original loop's trip count. This means all iterations of the loop body are meant to execute simultaneously, effectively removing the for-loop altogether.

6.2 Array Partitioning

HLS designs frequently include loops which perform the same operation on every element of an array or group of arrays. Arrays are typically synthesized into on-chip RAMs which have limited ports. These ports limit how many words can be read or written on a single cycle, typically 1 or 2 words per RAM depending on the primitive. If a loop that accesses one element per iteration is unrolled with factor N, then that array must be able to support accessing N words simultaneously. This is accomplished through array partitioning. Array partitioning with a factor of P splits the array into P separate RAMs. The number of words that can be accessed in one cycle is now multiplied by P. Array partitioning goes hand-in-hand with loop unrolling, as the additional ports are necessary to enable multiple iterations to read or write different elements of the array at the same time. Without array partitioning, the effects of loop unrolling would be extremely limited.

Array partitioning comes at the cost of increased resource usage for the fabric logic implementing the array access and the RAM primitives themselves. Partitioning can lead to wasted RAM space if the number of elements in one partition is smaller than the number of rows in each RAM primitive.

Just as loops can be fully unrolled, arrays can be completely partitioned. Completely partitioned arrays are synthesized as FFs rather than RAMs. This allows all elements to be read or written on the same cycle. Complete partitioning is typically used on smaller arrays, as it becomes too costly for larger ones.

6.3 Array Reshaping

Array reshaping is an extension of array partitioning in the sense that it enables multiple array elements to be accessed on the same cycle. By default, Vitis HLS will always place one element of an array in one row of a RAM, even if the row size is significantly larger than the data type. Scale-CNN places 16-bit float arrays into UltraRAMs which have 72-bit rows. By default, each 72-bit row would hold just 16 bits of data and the other 56 bits would go to waste. Array reshaping resolves this by commanding the tools to pack multiple elements into the same row. In this case, a reshaping factor of 4 is enough to make optimal use of the UltraRAMs. However, the reshaping factor can be even larger. If the reshaping factor were 8, the tools would horizontally concatenate UltraRAMs to form an effective RAM with 144-bit rows, enough to hold the 128 bits required.

6.4 Pipelining

Pipelining enables for-loops to break up the loop body into stages such that different iterations of the loop can run concurrently. All pipelines are characterized by two values: initiation interval (II) and iteration latency. The initiation interval indicates how many cycles the pipeline must wait between kicking off subsequent iterations, whereas the latency is the number of cycles it takes for one iteration to move through the entire pipeline. Ideally, pipelined loops have an II of 1, which means a new iteration can enter the pipeline on each cycle. Pipelining significantly reduces the total execution time of a loop when the trip count and latency are large. With a trip count of TC, an iteration latency of L cycles, and an initiation interval of II cycles, pipelining reduces the total loop execution time from $TC * L$ to $(TC - 1) * II + L$ cycles.

Pipelining and loop unrolling can be combined. A loop can be partially unrolled into multiple separate pipelines that run in parallel. However, as the trip count of the unrolled loop decreases (which happens has the unroll factor increases), the reduction in total loop latency begins to diminish. A fully unrolled loop should not be pipelined since its effective trip count is 1, so the latency will stay the same.

Pipelining is especially useful in nested loops. If the inner-most loop of a loop nest is pipelined, all outer loops in the nest will be merged with it to create one loop with a larger trip count. If an intermediate loop in the nest is pipelined, all loops beneath it are automatically fully unrolled. This is necessary so that the pipelined loop can achieve an II of 1.

6.5 Dataflow Pipelining

The pipelining described in the previous section is a low-level, fine-grained pipeline where each stage pertains to a small operation such as reading a RAM, or performing an addition. In contrast, dataflow pipelining is a higher-level, coarse-grained pipelining where each stage pertains to a task encapsulated in a function call. Dataflow pipelining has a number of key differences compared to regular pipelining:

- Each stage is a function rather than a single operation
- Stages can take multiple clock cycles
- Different stages have different latencies
- Flow control is decentralized, managed independently at the barriers between stages rather than each stage moving in lockstep.

The capabilities of the dataflow directive extend beyond pipelining. Dataflow supports different tasks running at different rates and supports any directed acyclic graph structure rather than being limited to simple feed-forward pipelines. It even has limited support for feedback between tasks. This allows it to implement any synchronous dataflow graph (SDFG), of which traditional pipelines are a subset. However, Scale-CNN does not utilize this capability.

6.6 Binding

Binding allows the user to specify an implementation for either a mathematical operation or storage of an array. Both of these have multiple options. For example, a floating-point addition can be implemented either entirely with LUTs, a single DSP and fewer LUTs, or two DSPs and even fewer LUTs. An array can be placed inside URAMs, BRAMs, distributed RAM, or FFs. Additionally, both operations and memories have configurable latencies. Vitis HLS uses internal heuristics to make these decisions on its own, but it cannot take any higher-level design goals into account. When the decisions made by the tool are not adequate for higher-level design goals, the user can override the tool's decisions through binding directives. Scale-CNN uses binding directives for different memories to properly balance URAM, BRAM, and FF usage for the overall design.

7 Cost Function

Scale-CNN evaluates multiple implementations for both layers and networks. As a consequence, we must define a cost function for quantitative comparisons of resource utilization.

For ASIC-based designs, total area is almost always used as the cost metric, represented in units of μm^2. Defining a cost function for FPGA designs, however, is not as straightforward because the physical hardware of the FPGA doesn't actually change between designs. HLS toolchains report utilization estimates of individual resources, but these quantities must be weighed together – a single DSP, for instance, is much more valuable than a single LUT. How to appropriately weigh the individual utilizations together is a subject of debate. Different works involving design space exploration for FPGA-based accelerators use different approaches.

Some works, such as LIN-Analyzer [19], simply ignore relative cost altogether and consider FPGA resources as "use it or lose it", where any design is considered equal as long as it does not exceed the available resources on chip [12]. This is perfectly acceptable when the design under consideration is the only IP that will be placed on the FPGA. For Scale-CNN, however, this is not appropriate because a network is composed of multiple layers and the tool needs to consider the relative costs of different layer implementations.

Other works attempt to weigh different resources by the relative areas of their underlying primitives. In [2], the costs of individual resources are expressed in terms of equivalent ALMs, or "eALMs" (an ALM is the basic unit of programmable fabric in Intel FPGAs.) However, using this kind of metric requires prior knowledge of the actual areas of the FPGA

primitives, which are not always publicly available.

Scale-CNN uses a sum-of-percentages cost, that can be expressed as follows:

$$Cost = \sum_i \frac{r_i}{R_i}$$

where i corresponds to each resource type, r_i is the quantity of that resource used by the design, and R_i is the quantity available on the FPGA. On Ultrascale+ FPGAs, there are five such resource types: FFs, LUTs, DSPs, BRAMs, and URAMs. This cost function is also used in [20]. The sum-of-percentages cost metric weighs the relative values of two units of different resources by the relative quantities of those resources on the FPGA. For instance, on the XCVU3P, a single DSP has the same cost as roughly 173 LUTs, as there are roughly 173x more LUTs than DSPs on chip. The downside of this metric is that it is FPGA-dependent. An SoC designer in the earliest stages of the design cycle may be trying to decide which FPGA is most appropriate by getting an idea of the range of resource costs for different design points. In this scenario, the cost metric is not helpful.

The "best" cost function will depend on the application. A designer may prefer a hybrid cost function that treats certain resources as "use it or lose it" while the rest use sum-of-percentages. Scale-CNN is designed to have a user-configurable cost function to be flexible to different circumstances.

8 Layer Design

Currently, Scale-CNN supports three layer types – a traditional convolution layer (CONV), a fused convolution-maxpool layer (CONV-MAX), and two convolution layers fused together (CONV-CONV). Scale-CNN was designed around Tiny Darknet which only involves convolution and maxpool layers until the very end, so it does not support other common layer types such as fully-connected. This is left for future work.

8.1 CONV Layers

The architecture of a basic CONV layer is shown in Figure 3. Each layer is implemented as a dataflow pipeline where each stage performs one step of the calculation for one or more output elements. At the beginning and end of each layer are the feature maps stored in ping-pong buffers implemented with URAMs. Between each stage are smaller buffers (either ping-pong buffers or FIFOs) implemented with BRAMs or FFs.

The feature maps arrays are always minimally reshaped with a factor of 4 to make optimal use of the URAMs. Each element is 16 bits while URAM rows are 72 bits wide. Reshaping with a factor of 4 tells the tools to pack four 16-bit words into the row. The feature maps may be reshaped with an even greater factor depending on how many elements need to be read out of it per cycle.

The first stage reads one $K * K * N$ convolution window of inputs as well as one or more filters. It simply copies the values into smaller BRAMs and does not perform any floating-point operations. The second stage multiplies the input fmaps with the weights to create products. The products are then sent into one or more accumulation stages to reduce the products into one sum per output element. The next stage is the "adjustment" stage

which applies batch normalization, bias, and activation. Although Tiny Darknet uses leaky ReLU activation, Scale-CNN only supports true ReLU to simplify the hardware and reduce resources. The final stage writes one or more output feature maps to the output URAMs.

8.2 Layer Implementations - Scaling

Scale-CNN generates multiple implementations for each layer. These implementations differ in their unroll factors used for the mid-level layer pipeline loop and low-level function pipeline loops. Each layer implementation is characterized by two different scale factors, one for each level.

The *read* and *multiply* stages are implemented as loops that iterate over $K * K * N$ input elements in a convolution window. These loops can be unrolled by a factor up to N, enabling multiple input channels to be processed simultaneously. For this reason, the unroll factor is called the *input channel scale factor*. While it is theoretically possible to unroll these loops further and parallelize the K dimensions, this is not done because kernel sizes are typically very small. Fully or almost-fully unrolling the *read* and *multiply* loops would result in low trip counts which would harm the resource efficiency of the low-level pipelines.

The second scale factor applies to the layer pipeline loop itself. This loop iterates over all elements of the output feature maps. With no scaling, each stage only works on computing one output element at a time. By unrolling this loop, we enable each stage to work on multiple output elements simultaneously. Note that this unrolling is not a true unrolling with pragmas since only certain parts of the loop body are duplicated. Just as the low-level pipelines can only be unrolled along the input channel dimension, this loop can only be unrolled along the output channel dimension. This means that all elements

computed simultaneously will always correspond to the same X/Y coordinate of the output. This is important because it means that all output elements can reuse the same convolution window of inputs, keeping the logic in the *read* stage simple. This scaling factor is called the *output channel scale factor*.

Both methods of scaling will reduce the overall layer latency at the cost of higher resources. Input channel scaling reduces the II of the layer pipeline while output channel scaling reduces its trip count. These methods of scaling can be combined together to create different implementations of the layer that vary in cost and performance. To keep the unrolling logic simple, maximize resource efficiency, and reasonably limit the number of layer implementations, both input and output channel scale factors must be factors of N and M, respectively. Scale-CNN generates one layer implementation for each permutation of the factors of N and M with a reasonable upper bound on the product of the two scaling factors (this prevents implementations that consume a very large amount of resources and are unlikely to be used when creating network implementations). The performance and cost of the implementation correlate with the product of the two scale factors. For the remainder of this paper, the terms *ICSF* and *OCSF* will be used for input and output channel scale factors, respectively.

Figure 4 shows an example of a CONV layer with an ICSF of 4 and OCSF of 2. The *read* stage still reads one convolution window of inputs, but now reads two filters instead of one. Additionally, the ICSF of 4 means the loops in both *read* and *multiply* stages process four elements per iteration in parallel. The multiplication and accumulations are duplicated, however, the *adjust* stage is not. This is because the *adjust* stage only needs to iterate over OCSF elements, which is typically small compared to the trip counts of the *read* and

multiply loops. The *adjust* stage is also implemented as a pipeline and will typically not be the bottlenecking stage in the layer pipeline except for extreme cases with very large ICSF and OCSF. In these extreme cases, the adjustment loop is partially unrolled to avoid bottlenecking the layer. Finally, the *write* stage writes 2 elements to the output feature maps per iteration.

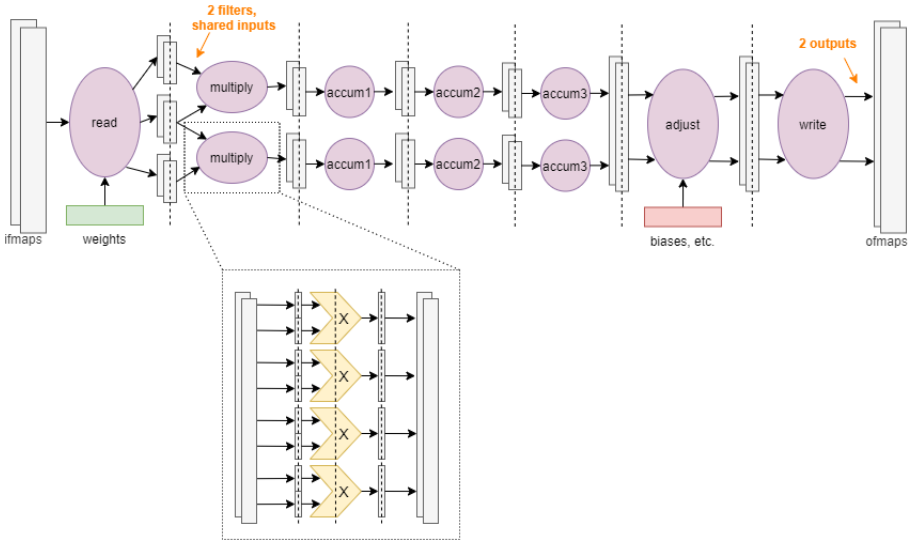

Figure 4: Scaled CONV layer with ICSF=4, OCSF=2, and three accumulation stages.

This method of scaling the performance for individual layers is very similar to what is proposed in [8], referred to as "inter-layer parallelism". However, there are some key differences. In their accelerator, iterating over the input channels is merged with the output channels in the mid-level layer loop rather than the low-level function loops. Their low-level loops only iterate over the kernel width and height dimensions, and are always fully unrolled. As a consequence, their layers are less flexible, and their cheapest implementations are likely costlier than those generated by Scale-CNN, since the kernel dimension loops are always unrolled.

8.3 Accumulation Stages

The accumulation stages of the layer pipeline reduce $K * K * N$ products into a single sum for each output element. Ideally, this would all be done in a single stage; however, this is usually impossible. While the *read* and *multiply* loops can be easily pipelined with an II of 1, accumulation cannot because a one-by-one accumulation with a single adder is an inherently serial operation. The accumulation can be parallelized with multiple adders, but each individual adder still accumulates serially, and their final sums must still be accumulated together at the end. This problem is exacerbated by input channel scaling which reduces the target II of the layer pipeline without changing the amount of accumulation that needs to be done, thereby placing stricter requirements on the accumulation to avoid bottlenecking the pipeline.

For these reasons, the accumulation is broken up into multiple distinct stages in the layer pipeline. Since different layer implementations will have different requirements in terms of maximum latency and number of products to reduce, Scale-CNN dynamically decides both the number of accumulation stages and implementations of each stage on a per-layer-implementation basis. For this reason, the generated C++ code actually differs between implementations of the same layer. Deciding how many and what types of accumulation stages to use can be framed as an optimization problem. However, instead of using a complex design space exploration approach, Scale-CNN instead uses a small number of guiding principles and heuristics to quickly make these decisions.

The first and most important principle is that the accumulation stages should never be the bottleneck in the layer pipeline. The reasoning for this is that adding additional stages to the pipeline will have a negligible impact on the total layer latency. Recall that

any pipelined loop's total latency can be expressed as $(TC - 1) * II + L$. For the layer pipeline, the trip count (TC) is equal to $OH * OW * M/OCSF$ (the number of output elements divided by the number of elements processed per iteration). The initiation interval (II) is just the latency of the longest stage, while the iteration latency (L) is the sum of all other stage latencies. II and L are typically on the same order of magnitude, and as a result, $TC * II \gg L$. Thus, even a substantial increase in the iteration latency by having multiple accumulation stages will have a negligible impact on the total layer latency due to the high trip count.

With this in mind, Scale-CNN estimates the II of the layer pipeline given the layer dimensions and the scale factors, and generates a number of accumulation stages that all have shorter latencies. To do this, it considers options between two extremes. On one extreme, it could aim for a small number of very resource-intensive stages that perform a high degree of reduction, while on the other extreme, it could opt for a larger number of cheaper stages that each perform fewer reductions. Additionally, it must decide if each stage's outputs should be stored in BRAMs (better for a large number of partial sums) or complete-partitioned registers (better for a small number of partial sums). This decision affects the read bandwidth of the next stage.

Scale-CNN attempts to strike a balance between these two extremes, using a number of hand-tuned heuristics that strive to achieve a minimum utilization of each adder (that is, how many additions each adder performs per stage execution). It chooses the accumulation stages one at a time and picks from four different types of accumulator architectures:

- *Simple Loop* – A single adder that accumulates one at a time, coded as a single for-loop

in C++. This is the cheapest accumulation stage possible. It is only used if it can reduce all inputs to a single output within the target latency. Therefore, it can only be used for the final accumulation stage.

- *Interleaved* – The interleaved accumulator can be thought of as multiple simple-loop accumulators that operate in parallel. If each step in an accumulation takes N cycles, having at least N accumulators allows the stage to dispatch inputs to different accumulators in a round-robin fashion without needing to wait for the previous addition to complete. The number of accumulators can even be greater than N so that multiple inputs are processed per cycle. This creates an effective pipeline with an II of 1 that has multiple outputs. Scale-CNN chooses the interleaved accumulator when the number of inputs is significantly larger than the read bandwidth. This is common for the first accumulation stage.

- *Pipelined Tree* – The pipelined tree accumulator connects multiple stages of parallel pipelined adders in a tree-like format. The number of inputs of the first stage, number of outputs of the last stage, and number of stages are all variable. Additionally, we must consider whether the inputs are stored in BRAMs, which limits how many words can be read per cycle, or registers, where the read bandwidth is unlimited. When the inputs are stored in BRAMs, the read bandwidth is fully utilized and the number of the inputs of the first stage equals the read bandwidth. The number of stages is then set as the maximum without exceeding the maximum latency of the accumulation stage. When the inputs are stored in registers, both the number of stages and words read per cycle are variable. In this case, the tool tries to find the option with the highest

degree of reduction that meets a minimum threshold in adder utilizations per function call. Pipelined tree stages are used whenever simple-loop and interleaved stages are not viable.

- *Unpipelined Tree* – Lastly, the unpipelined tree accumulator is used when the only option is a pipelined tree but the trip count of its encompassing loop would be 1. In this case, there is no point of wasting extra FFs and LUTs to create a pipeline, so the pipelining is removed. This is used as the final accumulation stage when the number of inputs is small but a simple-loop accumulator is still too slow.

8.4 Adjustment Stage

The adjustment stage applies batch normalization, bias, and activation to the fully-accumulated sums of products. The batch normalization and bias constants are stored together in an "adjustments" memory whose values are calculated off-line, similar to the weights. The stage is a simple pipeline that subtracts the mean, multiplies by the pre-calculated inverse square root of variance, adds the bias, and finally applies ReLU activation. Each sum is fed through this pipeline to create the final outputs. In rare cases with high OCSF and ICSF, this loop might need to be unrolled to avoid bottlenecking the layer pipeline. Although the ICSF is irrelevant to the stage's logic itself, higher values reduce the stage's maximum allowable latency to avoid becoming the bottlenecking stage. If the batch normalization is not desired for a stage, the mean and inverse square root values can be set to 0 and 1, respectively. One way this could be improved in the future is removing the batch normalization steps for layers where it is not needed.

8.5 CONV-MAX Layers

Rather than being their own independent stages, maxpool layers are fused with their previous convolution stage. This is an obvious optimization to exploit as the pooling logic is very simple and the memory savings are considerable. Table 2 shows that the memory requirements for Tiny Darknet feature maps reduce by 50% due to CONV-MAX fusing.

In CONV-MAX layers, the *write* stage is augmented with floating-point comparators and a running maximum for each output element. With a pooling factor of P, the stage only actually writes to the outputs every P^2 calls. When it writes to the output, the running maximum is reset. The order of iteration over the input feature maps is changed so that a full $P \times P$ block of inputs is processed before the next block.

8.6 CONV-CONV Layers

To enable further reductions of on-chip memory requirements, Scale-CNN has limited support for fusing two CONV layers together. Specifically, two CONV layers can be fused together only if the second one has a filter size $K = 1$. As discussed in [1], fusing two CONV layers with $K > 1$ together requires careful consideration of the pyramid-shaped data dependencies between them. Consider an example where $K = 3$ for both layers. In a naive approach, the first layer would need to process a full 3x3 window of outputs for each output of the second layer, resulting in a 9x increase in work. A more intelligent approach would be to exploit data reuse through a sliding window buffer of intermediate feature maps between the two layers. Implementing this within the context of a dataflow pipeline is possible but not trivial, and is not currently supported by Scale-CNN. Instead, Scale-CNN requires that the second layer have 1x1 filters. This limits it to the simple case where the

production rates of the two layers are 1-to-1 and a sliding window buffer is not needed.

Figure 5 shows the design of a CONV-CONV layer. All stages for both layers are connected in a single dataflow pipeline, broken up into two parts (L1 and L2). The stages pertaining to the first layer work identically to ordinary CONV layers. There are still the two scaling factors, ICSF and OCSF which specifically apply to the L1 stage. The output of the L1 adjustment stage are a $1 \times 1 \times OCSF$ array of "intermediate" feature maps between the two layers.

The key difference between the L1 and L2 stages is that the L2 stages do not have access to all their input channels at once. The L2 stages turn a $1 \times 1 \times M_1$ array of inputs into a $1 \times 1 \times M_2$ array of outputs, where M_1 and M_2 are the output channels for L1 and L2 respectively. However, the L1 stages only produce a $1 \times 1 \times OCSF$ subset of those inputs per iteration, where $OCSF$ is an integer factor of M_1. As a consequence, the L2 stages break up the computation of the $1 \times 1 \times M_2$ outputs over $M_1/OCSF$ iterations, only writing to the output RAMs on the last. Whereas L1 only calculates $OCSF$ output channels at a time, L2 calculates the output element for all M_2 output channels simultaneously. This is balanced out by only receiving a few inputs at a time.

The *read* stage of L2 is merged with the *multiply* stage given the simplicity of the layer's dimensions. Unlike L1 and other CONV layers, the L2 stage only has a single accumulation stage. Whereas the L1 accumulation stages reduce all inputs to a single sum, the L2 accumulation stage instead receives M_2 independent groups of $OCSF$ products and reduces them to M_2 sums. These sums are then fed into the final stage, which is shown in detail in Figure 5. This stage maintains running sums for each of the M_2 outputs that retain their values across function calls through use of `static` variables. In a pipelined manner,

the stage will add the new partial sums to the running sums. If it is the final iteration for computing a $1 \times 1 \times M_2$ array of outputs, it sends all of the complete sums through the adjustment pipeline where they are finally written to the output RAMs.

Table 2 shows that CONV-CONV layer fusing with Tiny Darknet reduces the feature map memory requirements by 44% compared to only using CONV-MAX fusing. Both fusing techniques combined reduce the feature map memory requirements by 72%. Tiny Darknet's CONV layers alternate filter sizes of 1 and 3 which make it very amenable to Scale-CNN's limitations – however, this might not hold true for other networks.

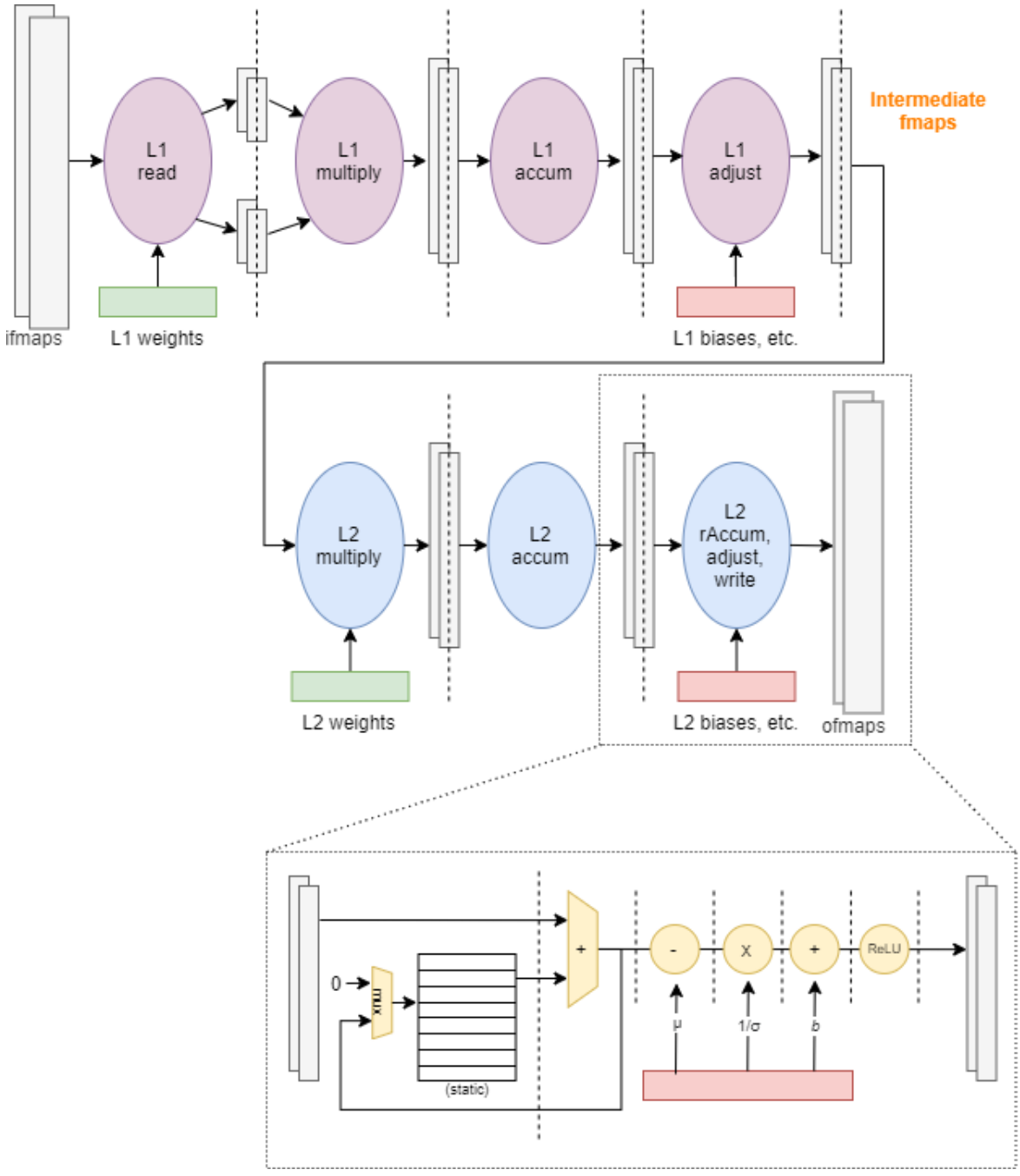

Figure 5: Architecture of a fused CONV-CONV layer.

9 Layer Implementation Results

Figure 6 shows the cost and performance of each possible implementation for a CONV-MAX layer. As expected, the points exhibit a tradeoff between the two metrics. Each point's location on the curve roughly correlates to the product of its two scale factors. Points with an equal product of scaling factors are close to each other on the graph. Note that there is not a point for every possible combination of scale factors. Scale-CNN places a reasonable upper limit on the product of the two scale factors to prevent any single layer from consuming too many resources.

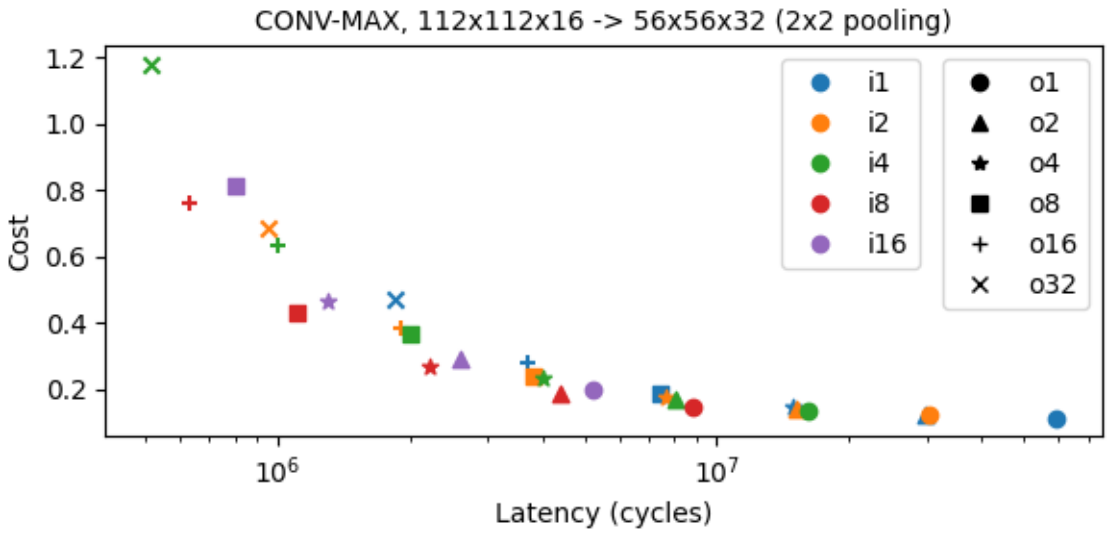

Figure 6: Cost and performance for different implementations of a CONV-MAX layer

On the bottom-right is the design point *(i1, o1)* where both scale factors are 1. This is the slowest, cheapest implementation of the layer where no loops are unrolled. The point on the opposite extreme has the most unrolling with $ICSF = 4, OCSF = 32$. One would expect that this point should have a roughly 128x speedup over the point with no

scaling. In reality, it is only about 91x faster. The difference can be attributed to the fact that unrolling the low-level loops (input channel scaling) scales down their trip counts but not their latencies. Specifically, having $ICSF > 1$ will reduce the total loop latency from $(TC - 1) * II + L$ to $((TC/ICSF) - 1) * II + L$. Therefore, the speedup is deteriorated by the pipeline's latency. As the ICSF increases, the gap between the ideal and actual speedup widens as the pipeline latency becomes a larger percentage of the total loop latency. In contrast, output channel scaling barely suffers from this problem because the layer pipeline latency is tiny compared to the total loop execution time, as mentioned in Section 8.3. The point *(i1, o32)* has an almost perfect 31.996x speedup over *(i1, o1)*.

However, this does not mean that output channel scaling is always better or more efficient than input channel scaling. In fact, we can see that point *(i1, o32)* is not even Pareto optimal; point *(i8, o8)* is both faster and cheaper. The reason for this is debatable. A direct cost comparison between two close design points is not necessarily meaningful because (1) the cost is derived from utilization estimates from the HLS software that may not be entirely accurate, and (2) the cost function given in Section 7 is an oversimplification of resource usage. A different cost function, or the same cost function on another FPGA, may indicate the opposite relation between the two points. Nonetheless, a general trend observed among the design points for multiple layers is that points where both scale factors are greater than 1 are more likely to be Pareto optimal than points where one scale factor is large and the other is 1.

10 Network Design

As introduced in Section 4 and shown in Figure 3, the network is built by connecting all of its layers in another higher-level dataflow pipeline where each layer is a separate stage separated by ping-pong buffers holding feature maps implemented with URAMs. Each layer simultaneously works on a different inference at a time, enabling the network to work on multiple inferences simultaneously. The II of this pipeline is dictated by the total latency of the slowest layer. For Tiny Darknet, the II is on the order of 100s of thousands to 10s of millions of cycles depending on the exact implementation. The inference throughput of the network (inferences per second) can be calculated as the clock frequency (cycles per second) divided by the II (cycles per inference).

10.1 Layer Selection

Just as Scale-CNN generates multiple implementations for each layer, it also generates multiple implementations for each network. Different network implementations are generated by "mix-and-matching" different layer implementations together. Scale-CNN first synthesizes every implementation of every layer in the network and analyzes each one's cost and latency. It then discards all implementations that are not Pareto optimal. Once it has all Pareto optimal options for each layer, it generates multiple implementations for the network by choosing different combinations of options for each layer.

Although each layer implementation synthesis is relatively short (typically 2-3 minutes with Vitis HLS 2020.2 on a 2.5 GHz Intel Xeon E5-2640 CPU), there can be hundreds of implementations across all layers in the network. Synthesizing all possible implementations of all layers can take many hours. One way to mitigate this long time is to eliminate layer

implementations that will never be chosen. This is shown in Figure 7, which overlays the Pareto optimal design points of the first four layers of Tiny Darknet. In a hypothetical network that consists only of these four layers, the dotted line indicates the smallest possible II of the network pipeline, as this is the slowest of the fastest options for each layer. The fastest possible network implementation would use the design point on this line and the first design point to the left of this line for all other layers. Beyond these points, any faster layer option would not speed up the network pipeline, since it would still be bottlenecked by the layer on the dotted line. Scale-CNN estimates the latencies of all layer implementations before synthesis, identifies these points, and discards them to save time.

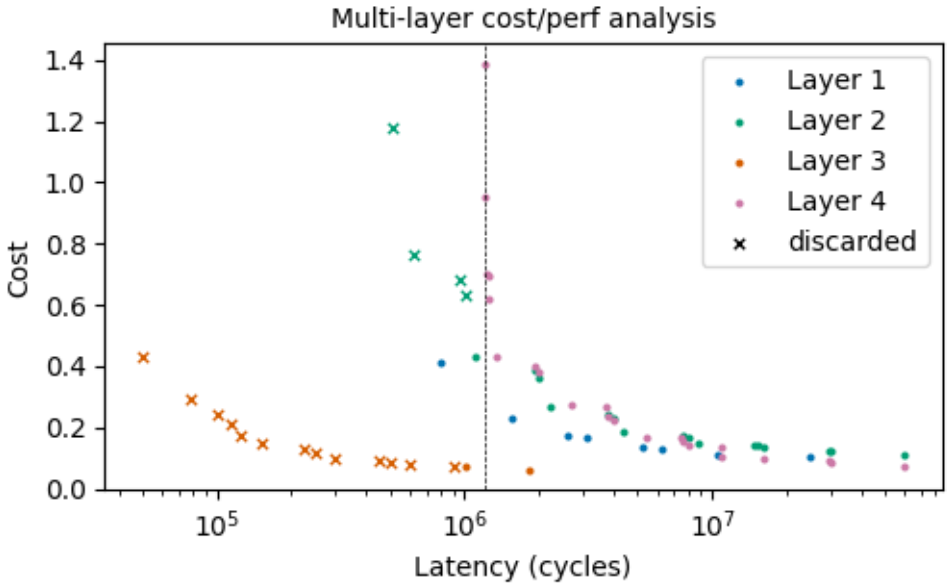

Figure 7: Performance and cost for all implementations of the first four layers of Tiny Darknet with layer fusing.

One of the primary objectives of Scale-CNN is to create designs that maximize the

efficiency of utilized resources. For the overall network, the slowest layer is the weakest link and will limit the resource efficiency of all other layers. During inference, each layer that is not the slowest will spend a certain amount of time idling while waiting for the slowest layer to finish. Choosing a fast, expensive implementation of a layer is a waste of resources if that layer is not the "critical path" of the network pipeline, since making the layer faster does not improve the pipeline's throughput. Keeping this in mind, Scale-CNN chooses layers with the goal of making the layer latencies as balanced as possible. By balancing the latencies, we maximize the percentage of time that all layers are doing useful work. Scale-CNN uses a simple iterative algorithm to generate network implementations given a list of implementations for each layer:

1. Start by choosing the slowest, cheapest implementation of each layer. This becomes the first implementation of the network and is the cheapest one possible.

2. Identify the layer with the highest latency. This is the bottlenecking layer, and its latency is the II of the network pipeline. There may be multiple layers with the same latency.

3. Switch to the next fastest implementation for the bottlenecking layer, leaving all other layers the same. In the event that multiple layers have the exact same bottleneck latency, make all of them faster. This becomes a new implementation for the network and will have a smaller II.

4. Repeat steps 2-3 until the bottlenecking layer is already using its fastest implementation and cannot be sped up any further. At this point, we cannot reduce the network pipeline's II any further.

This approach is analogous to the method of increasing the maximum clock frequency of a synchronous digital circuit – find the critical path, shorten it until it's not the critical path, and repeat.

Figure 8 shows the layer latencies of four different implementations of Tiny Darknet with layer fusing. The leftmost one is the one with the slowest implementation for each layer. This option has a relatively high variance between the layer latencies and thus suffers from poor resource efficiency; however, it is nonetheless the cheapest network implementation possible and thus guaranteed to be Pareto optimal. The three layers whose latencies are far below the rest are unfused CONV layers with 1x1 filters. Given the nature of the network dimensions, these layers have a relatively small amount of work to do compared to those with 3x3 filters. For this reason, even their slowest implementations are very short.

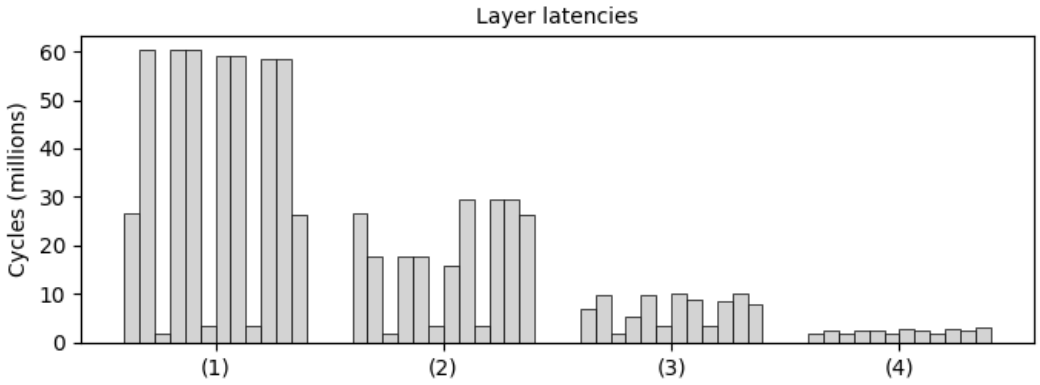

Figure 8: Layer latencies for four different implementations of Tiny Darknet with layer fusing: the slowest possible (1), the fastest possible (4), and two in the middle (2,3)

As the layers latencies decrease, the II of each new implementation (indicated by the height of the highest bar) decreases along with the variance of the latencies. This means that as the network implementations get faster, the resource efficiency naturally increases.

In practice, the fastest implementations of the network are likely to exceed the available resources on the FPGA in at least one category. Scale-CNN totals up the estimated resource requirements of all layers and discards network implementations that are estimated to exceed 100% utilization of any single resource type. This is how Scale-CNN is able to "scale" its generated designs to different FPGAs – it will only provide design points that can fit inside the FPGA specified. This is convenient to the designer, since by simply switching to a larger FPGA, Scale-CNN will automatically find new, faster design points that take advantage of the additional resources.

10.2 Memory Selection

The limitations in on-chip memory require Scale-CNN to carefully decide which arrays are placed in BRAMs vs. URAMs, which can be controlled by binding directives. The first approach was to place all feature maps in URAMs and all other arrays in BRAMs. While this worked decently well, it occasionally created problems with over-utilization of BRAMs. A common pattern seen in CNNs (including Tiny Darknet) is that the earlier layers have large widths/heights and very few channels, while later layers have the opposite. This means that as you get further down the network, the amount of memory requirements to store all filters increases. This trend can be seen in Table 2.

Although the FPGA used in the experiments has enough BRAMs to hold all filters for Tiny Darknet, some layers would require several hundred BRAMs to hold all of their filters. This leads to an imbalance between BRAM and URAM utilization, and risks running out of BRAMs if the design is ever integrated in a higher-level system. To mitigate this, Scale-CNN dynamically chooses the RAM types of filters on a per-layer basis. First, it plans to place

all filter arrays in BRAMs. It then estimates the utilization percentages of BRAMs and URAMs for the entire design. If a significantly higher percentage of BRAMs will be used compared to URAMs, it finds the layer with the highest total filter size that is currently set to BRAMs and switches it to URAMs instead. Then it recalculates and repeats until the estimated BRAM and URAM percentages are as balanced as possible.

11 Network Implementation Results

11.1 Experimental Setup

Using Scale-CNN, we generated network implementations for Tiny Darknet on the smallest Virtex Ultrascale+ FPGA (XCVU3P) under three configurations: without CONV-CONV fusing, with CONV-CONV fusing, and with CONV-CONV fusing in addition to a faster clock period (6 ns compared to 10 ns). Each network implementation's cost was estimated as the sum of the costs of the individual layer implementations. The results are shown in Figure 9.

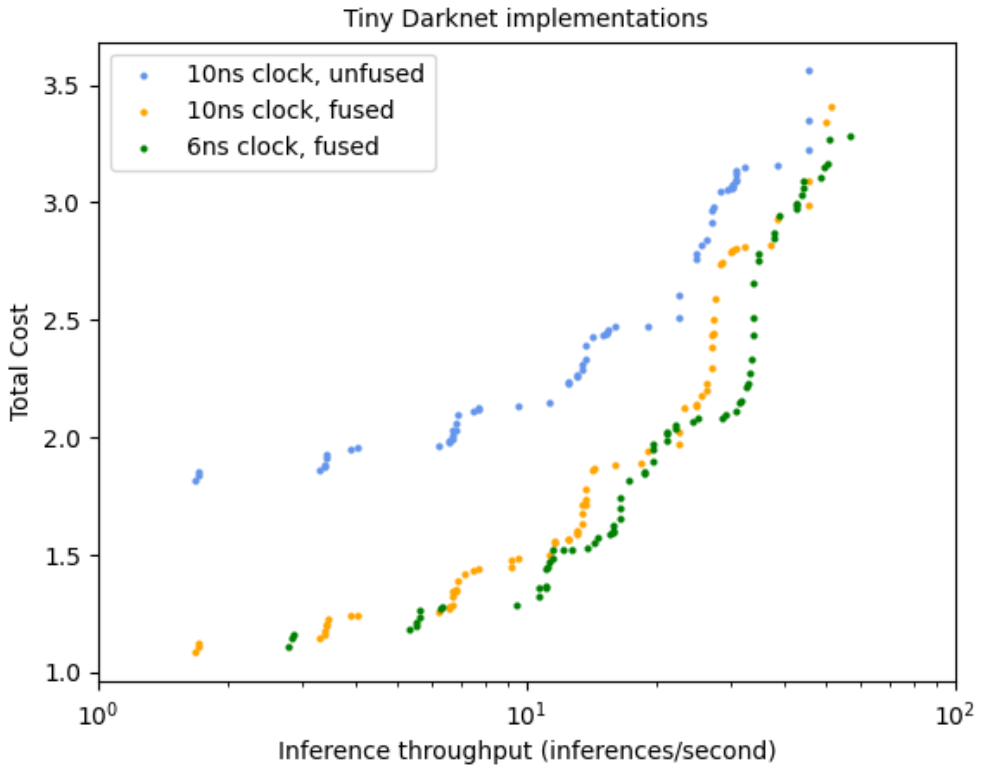

Figure 9: Cost and inference throughput for different implementations of Tiny Darknet

11.2 Cost/Performance Analysis

The data in all three cases shows a noticeable trend of a stair-shaped progression along the cost/performance tradeoff curve. This can be explained by the fact that layer latencies across different layers tend to be tightly clustered together. An example of this can be seen in implementation (1) in Figure 8 – the slowest layer's latency is very close to six other layers. The network as a whole will not become significantly faster until all six of these layers are made faster. As Scale-CNN selects the faster implementations for these layers one at a time, the cost of the overall network increases with little to no decrease in inference throughput. Once the last of the layers have been sped up, the II drops abruptly with only a slight increase in cost, creating the stair-shaped trend seen in the graph. The tight clustering of layer latencies can be attributed to regularities and symmetry in the layer dimensions.

Comparing the unfused network against the fused network, we can see that the fused network's design points are always superior. None of the design points of the unfused networks are Pareto optimal. This is expected since the reduced memory requirements of CONV-CONV fusing vastly reduce the cost of the overall network but the layer latencies are largely the same.

Comparing the fused networks with different clock frequencies, we can see that the Pareto frontier (that is, the set of all Pareto optimal points) mostly consists of points with the 6 ns clock but does include a few points with the 10 ns clock. This shows that a faster clock is not always necessarily better than a slower one. With a faster clock period, the HLS tools schedule the individual operations differently and will use more logic to satisfy the stricter timing requirements. Additionally, shorter clock periods often lead to longer

pipeline latencies since the tools are not able to schedule all operations of the loop body in the same number of cycles. The time penalty incurred by having these longer latencies may outweigh the time saved by using a faster clock.

11.3 Design Point Recommendations

In addition to generating multiple design points for a network, Scale-CNN is capable of recommending a subset of those points as most likely to be of interest to the designer. When implementing Tiny Darknet on the XCVU3P, there are a few dozen design points to choose from. There are many more theoretical design points that are discarded as the FPGA does have enough resources for them. On a larger FPGA, these points would fit and the number of possible design points could easily exceed 100.

These recommendations are made on the basis of the stair-shaped trend in design points seen in Figure 9. Even among the Pareto optimal points, the designer is unlikely to choose a point if there is another point that is significantly cheaper for only a small decrease in throughput. Scale-CNN finds these points that provide the best "bang-for-your-buck" by using simple K-means clustering. It clusters the network design points by II and recommends the cheapest design point in each cluster. The difference between IIs tends to shrink as the design points become faster, so the exact clustering metric used is $log(II)$ to prevent skewing the cluster centers towards the slower design points. The recommended design points are shown in Figure 10.

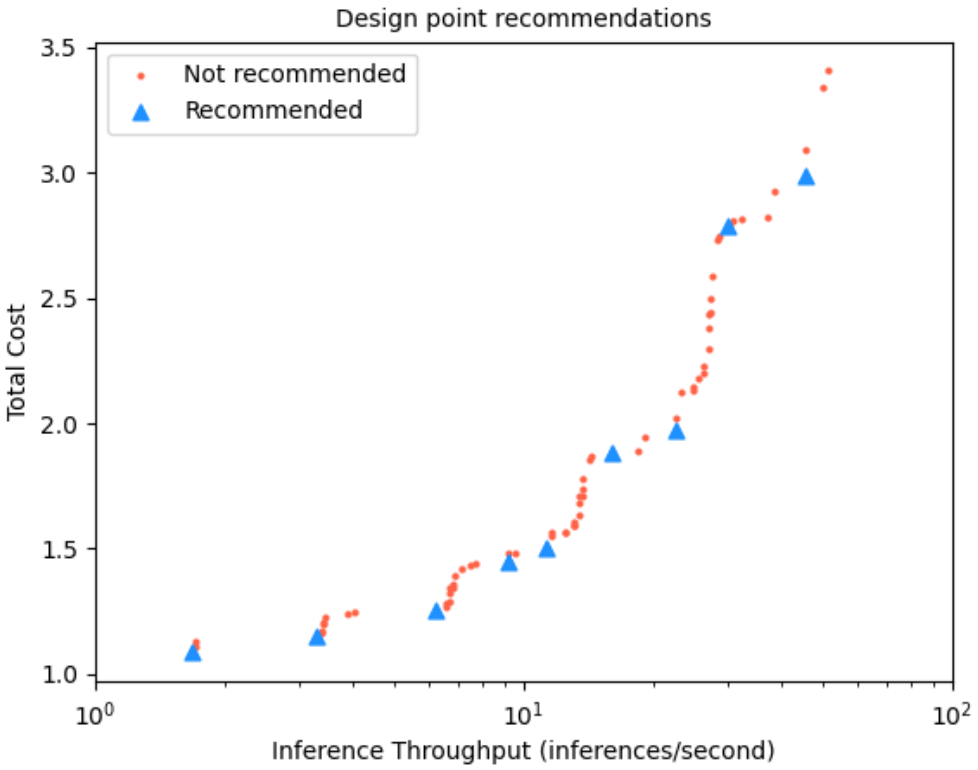

Figure 10: Recommended design points for Tiny Darknet with fusing

12 Future Work

Scale-CNN is still in its infancy stage and could be extended in many ways. Most importantly, it could be extended to support more layer types (such as fully-connected, softmax, and element-wise) and have configurable batch normalization and activation functions. This would be the first step in enabling it to support other networks besides Tiny Darknet.

Another vital improvement would be extending the tool's layer fusing capabilities. As mentioned in Section 5.6, Scale-CNN's architecture naturally requires a considerable amount of on-chip memory. Even for small networks such as Tiny Darknet, the design can only fit on the largest of FPGAs when layer fusing is used to mitigate these requirements. Further cutting down the memory requirements would make it feasible to create designs for larger networks, as well as make designs for smaller networks on relatively smaller FPGAs such as the Kintex Ultrascale+ family. This could be done in two ways. First, it could be extended to support fusing CONV-MAX and CONV layers together with the same limitations that currently exist. This would enable further layer fusing for Tiny Darknet, but may not be useful for other networks. The second and more challenging task would be to extend it to support CONV-CONV fusing in the general case, where there is no limitation that the second layer have 1x1 filters. This could be done by having an additional stage that maintains a sliding window buffer between the two layers, enabling data reuse of the first layer's outputs as the sliding window moves.

Lastly, Scale-CNN designs should be fully synthesized and implemented on a physical FPGA integrated with a higher-level CPU/FPGA coprocessing system. This would allow performance and power comparisons to related works, as well as analysis on the accuracy of HLS utilization estimates. This could lead to improved cost functions that better reflect

the true utilization numbers of final designs – anecdotal evidence suggests that FF and LUT utilizations are considerably overestimated by the HLS toolchain.

13 Conclusion

In conclusion, we introduce Scale-CNN, a tool for generating high-throughput CNN accelerators on FPGAs. We suggest a scalable network pipeline architecture where each layer is a separate stage working on separate inferences simultaneously. By storing all feature maps and weights on chip, we eliminate the need for off-chip memory accesses for hidden layers, which are a common bottleneck for other CNN accelerators. Previously, this was infeasible due to the considerable memory requirements for storing all feature maps and weights on-chip. However, reducing these memory requirements through layer fusing and use of FP16 data makes this architecture possible for small to medium-sized networks on modern high-end FPGAs with 10s to 100s of Mb of on-chip storage.

Whereas other CNN accelerator works either provide a single design or a parameterized design that must be manually tuned by the designer, Scale-CNN generates several design points that span a wide range on the cost/performance tradeoff curve and are all Pareto optimal. It does this by generating multiple implementations for each layer, then "mix-and-matching" them to create different implementations of the network. All design points make efficient use of hardware resources by using a three-level hierarchical pipelined loop structure. Different network implementations primarily differ in the unroll factors of the mid-level and low-level pipeline loops, which correspond to parallelizing output channels and input channels, respectively. High resource efficiency is acheived by balancing the stage latencies of the high-level and mid-level dataflow pipelines and making sure the low-level function pipelines have an initiation interval of 1.

We used Scale-CNN to generate accelerator implementations of Tiny Darknet on the XCVU3P FPGA. The tool generated implementations with inference throughputs ranging

from roughly 1 to 56 inferences per second using a combined percentage of 22% to 66% of the FPGAs resources. We then compared implementantions for three different Tiny Darknet configurations. First, we compared the designs with layer fusing against those without and found that layer fusing provides considerable cost savings without a significant hit in performance. Second, we compared two fused networks with different clock speeds and found that faster clock speeds are not always better due to the increased cost from meeting stricter timing requirements.

Although Scale-CNN is designed to support any convolutional neural network, its current feature set only supports Tiny Darknet. With further development, Scale-CNN could become a useful tool for generating high-throughput CNN accelerators in applications where using large high-end FPGAs is possible. To encourage this, we have published the tool for further contributions at `https://github.com/dlrauch15/scale-cnn`.

Appendix

#	Type	Input Dims			Filter size	Filter storage (Mb)	Input fmaps storage (Mb)		
		H	W	C			no fusing	conv-max	conv-max / conv-conv
0	conv	224	224	3	3	0.007	4.59	4.59	4.59
1	max	224	224	16	-	-	24.50	**0**	**0**
2	conv	112	112	16	3	0.07	6.13	6.13	6.13
3	max	112	112	32	-	-	12.25	**0**	**0**
4	conv	56	56	32	1	0.008	3.06	3.06	3.06
5	conv	56	56	16	3	0.28	1.53	1.53	1.53
6	conv	56	56	128	1	0.03	12.25	12.25	**0**
7	conv	56	56	16	3	0.28	1.53	1.53	1.53
8	max	56	56	128	-	-	12.25	**0**	**0**
9	conv	28	28	128	1	0.06	3.06	3.06	3.06
10	conv	28	28	32	3	1.13	0.77	0.77	0.77
11	conv	28	28	256	1	0.13	6.13	6.13	**0**
12	conv	28	28	32	3	1.13	0.77	0.77	0.77
13	max	28	28	256	-	-	6.13	**0**	**0**
14	conv	14	14	256	1	0.25	1.53	1.53	1.53
15	conv	14	14	64	3	4.50	0.38	0.38	0.38
16	conv	14	14	512	1	0.50	3.06	3.06	**0**
17	conv	14	14	64	3	4.50	0.38	0.38	0.38
18	conv	14	14	512	1	1.00	3.06	3.06	**0**
19	conv	14	14	128	1	1.95	0.77	0.77	0.77
20	(final)	14	14	1000	-	-	5.98	5.98	5.98
					Total:	**15.819**	**110.11**	**54.98**	**30.48**

Table 2: On-chip memory requirements of Tiny Darknet with FP16 activations and weights

Bibliography

[1] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-Layer CNN Accelerators. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[2] A. Boutros, S. Yazdanshenas, and V. Betz. You Cannot Improve What You Do Not Measure: FPGA vs. ASIC Efficiency Gaps for Convolutional Neural Network Inference. *ACM Transactions on Reconfigurable Technology and Systems*, 11(3):1–23, 2018.

[3] 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*, pages 269–284, 2014.

[4] Y. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263, 2016.

[5] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1, 2016.

[6] L. Ferretti, G. Ansaloni, and L. Pozzi. Lattice-Traversing Design Space Exploration for High Level Synthesis. In *IEEE 36th International Conference on Computer Design*

(ICCD), pages 210–217, 2018.

[7] L. Gong, C. Wang, X. Li, H. Chen, and X. Zhou. MALOC: A Fully Pipelined FPGA Accelerator for Convolutional Neural Networks With All Layers Mapped on Chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2601–2612, 2018.

[8] C. Huang, S. Ni, and G. Chen. A Layer-based Structured Design of CNN on FPGA. In *IEEE 12th International Conference on ASIC (ASICON)*, pages 1037–1040, 2017.

[9] H. T. Kung. Why Systolic Architectures? *Computer*, 15(1):37–46, 1982.

[10] J. Redmon. Darknet: Open Source Neural Networks in C. `http://pjreddie.com/darknet/`, 2013–2016.

[11] S. Saha. A Comprehensive Guide to Convolutional Neural Networks - the ELI5 way. `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`.

[12] B. Schafer and Z. Wang. High-Level Synthesis Design Space Exploration: Past, Present, and Future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2628–2639, 2020.

[13] V. Sze, Y. Chen, T. Yang, and J. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[14] Y. Umuroglu, N. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference.

In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 65–74, 2017.

[15] W. Wulf and S. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[16] Xilinx. Vitis High-Level Synthesis User Guide. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf`.

[17] W. Xuechao, C. Hao Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.

[18] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 161–170, 2015.

[19] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar. Lin-Analyzer: A High-level Performance Analysis Tool for FPGA-based Accelerators. In *53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[20] G. Zhong, V. Venkataramani, Y. Liang, T. Mitra, and S. Niar. Design Space Exploration of Multiple Loops on FPGAs using High Level Synthesis. In *IEEE 32nd International Conference on Computer Design (ICCD)*, pages 456–463, 2014.

Vita

Daniel Levi Rauch was born in 1992 in Houston, Texas. He attended Vanderbilt University in Nashville, Tennessee for his undergraduate studies and graduated in 2015 with a Bachelor of Engineering degree in Computer Engineering. He then moved to Austin, Texas where he worked as a FPGA digital hardware engineer for NI (formerly National Instruments) for four years. In 2019, he left to pursue a Master's degree at the University of Texas at Austin in Electrical and Computer Engineering with a focus on computer architecture. He can be reached at daniel.rauch@utexas.edu

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.