From TensorFlow Graphs to LUTs and Wires: Automated Sparse and Physically Aware CNN Hardware Generation

Mathew Hall* and Vaughn Betz*†
*Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada
†Vector Institute, Toronto, Canada
Email: mathew.hall@mail.utoronto.ca, vaughn@ece.utoronto.ca

***Abstract*—We present algorithms and an architectural methodology to enable zero skipping while increasing frequency in per-layer customized data flow Convolutional Neural Network (CNN) inference accelerators for FPGAs. Data flow architectures leverage the static configurability of FPGAs to increase processing efficiency, reduce dynamic muxing, and save routing wires. While this holds out the promise of high efficiency, these architectures require a different circuit to implement every CNN, making automated exploration and implementation of the accelerator essential. Each accelerator has layer-specific subcircuits with CNN-specific parallelization parameters and CNN graph topology-based interconnection that impact fanout and routing congestion, which lower the hardware operating frequency with naive implementation strategies. To address this, we designed latency insensitive hardware templates that build a model of signal fanout and span and instantiate different structures within each compute unit to ensure a high operating frequency regardless of CNN topology and parallelism settings. We also leverage the hardware efficiency of data flow architectures to add support for zero-weight-skipping at a normalized area cost less than one half of prior work. The overall optimization tool chooses parallelism settings for each layer-specific hardware unit to balance throughput across all layers of the CNN, while respecting the FPGA device limits on available buffering space and DSP blocks. Together these optimizations enable throughput on a sparse Resnet-50 model at a batch size of 1 of 4550 images/s, which is nearly 4x the throughput of NVIDIA's fastest machine learning targeted GPU, the V100, and outperforms all prior work on FPGAs.**

I. Introduction

Recent work has shown that we can maintain accuracy and prune nearly 90% of the weights from neural networks [1]. If computation of the pruned parameters can be skipped with custom hardware accelerators, we can potentially realize latency and throughput improvements of up to 10x; however, the pruning process makes the data layout irregular and more challenging to efficiently accelerate. Recent works have tried to overcome this challenge by taking advantage of the flexible logic, routing, and memories available on FPGAs, but works targeting sparse Convolutional Neural Networks (CNNs) have been limited by low multiplier utilization and some inefficiencies in mapping certain layers to their hardware architecture [1].

Layers in a neural network each have a different set of properties (e.g. input dimensions, stride, kernel size, etc.) that make it challenging to optimize one type of Processing Element (PE) that is efficient for all of them, and the less regular structure of sparse CNNs heightens this challenge. For example, a convolutional layer with a kernel size of 7×7 in a neural network with 50 layers imposes a requirement that the PEs must be able to handle kernels that large, even if all of the other layers only use kernels of size 3×3. During most of the execution time, any additional hardware to support those larger kernels is underutilized. While not all architectures pay a substantial cost for variable kernel size specifically, they likely pay some penalty for some set of parameters that vary through the network. Most FPGA and ASIC accelerators use a PE architecture that overprovisions hardware in each PE to handle a variety of layer types.

In this paper we present a novel accelerator architecture that solves these problems by tailoring hardware specifically to each neural network layer and which naturally supports weight sparsity. We then pipeline across network layers to achieve high throughput and low latency. Since building custom PEs for each layer of every CNN architecture manually would be intractable, we also developed a network compiler that accepts a TensorFlow graph as input, performs per-layer optimizations, and produces a synthesizable verilog accelerator that implements the network. We integrate this with a PCIe core and validate network accuracy and accelerator throughput in physical hardware.

The contributions of this paper are as follows:

- The novel HPIPE architecture that allows layer-specific optimizations and dramatically reduces the amount of soft logic required to implement 0-weight skipping in convolutional neural networks.
- An automated flow that converts TensorFlow network models directly to an optimized HPIPE hardware implementation.
- An evaluation of the HPIPE architecture on a sparse ResNet-50 and two variants of MobileNet that both demonstrate the efficiency of our approach and yield higher throughput at batch size 1 than any FPGA or GPU solutions known to the authors.

II. Background and Related Work

A. Convolutions

There are three types of 2D convolutions discussed in this work. The most basic is a 2D convolution that operates on a 3D tensor and outputs a 3D tensor. The weights have 4 dimensions,

$k_h \times k_w \times c_i \times c_o$, where k_h is the height of the weight tensor, k_w the width, c_i the size of the z dimension of the 3D input tensor, and c_o the size of the z dimension of the output 3D tensor. At each (x,y) point in the input image c_o slices of the kernel with shape $k_h \times k_w \times c_i$ are multiplied element-wise and reduced by summation to produce c_o output points.

The second type of convolution is a pointwise 2D convolution, which is a special case of the standard 2D convolution where $k_h \times k_w = 1 \times 1$.

The final type is what is called a depthwise convolution. In contrast to the basic convolution, the kernel is of shape $k_h \times k_w \times c_i \times n$ where n is a channel multiplier that is typically 1. Also in contrast to the basic 2D convolution, the summation following the element-wise product reduces along only the x and y dimensions, so while there is no c_o, the number of channels is preserved (or multiplied by the channel multiplier).

B. Sparsity and Weight Pruning

Weight pruning is the process of removing (assigning to zero) unimportant weights in a trained neural network. Recent works have shown that as many as 90% of the weights can be removed without impacting classification accuracy [1, 2]. This offers the opportunity to save memory and memory bandwidth by storing compressed weights, and also the opportunity to skip ineffectual multiplications by pruned (zero) weights.

C. Related Work

Many accelerators targeting FPGAs have taken advantage of sparsity for Fully Connected (FC) or Long Short Term Memory (LSTM) units [3, 4, 5, 6]. But fewer have taken advantage of sparsity in convolutional layers. FC layers and LSTMs have no weight re-use and are memory bound to begin with, so weight pruning provides both a computational and memory bandwidth reduction. By contrast convolutions share many weights so while we can reduce the computational requirements by pruning, the memory bandwidth reductions are less significant and come with an added cost of less regular computation and lower activation re-use. The authors are aware of two other attempts to accelerate sparse convolutions on FPGAs.

In [7], Kung et al. used a novel technique to prune weights and subsequently compress them back into dense weights by combining columns of non-overlapping weights together. The result was a very efficient accelerator, but the technique only works for pointwise convolutions. They also implemented a shift unit that implements something similar to a depthwise convolution, but their accelerator can only support one very specific type of neural network.

In [1], Lu et al. developed a PE-based sparse CNN accelerator that handles a wider variety of convolutions than [7]; however, their performance was limited by a lower frequency than comparable dense accelerators (200MHz), poor mappings of particular layers to their PEs, and a low DSP utilization of only 45% for an application that is primarily multiplication-bound.

A number of sparse CNN accelerator ASICs have been proposed [8, 9, 10]. In this paper we qualitatively assess some of the high level architectural features of SCNN [9] and determine they do not translate well to FPGAs. NullHop [10] also provided FPGA device utilization numbers (they validated their design on an FPGA) which show that they consume 83% of the logic while using only 6.3% of the DSPs and run at a frequency of only 60MHz. The zero-weight skipping architecture of Lu et al. [1] finds a more efficient solution for FPGAs, but their soft logic utilization still limits them to only 45% DSP utilization.

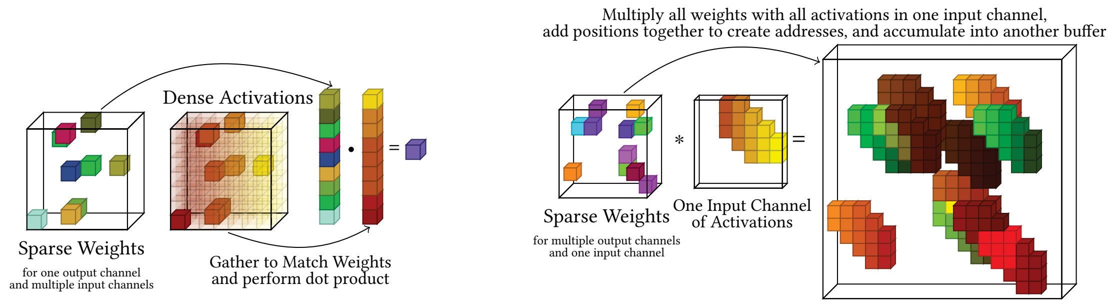
Our approach differs from prior work in how it leverages the hardware of FPGAs. While other works have proposed generic PEs that are used for all layers, our work dedicates hardware to every component of an input neural network. This better leverages the programmable routing and logic resources of FPGAs and enables DSP utilization of 87% on a sparse Resnet-50 model and 89% on a dense MobileNet-V1 model.

III. Architectural Choices

Convolutions are the most computationally expensive operation in a typical CNN. Other works have reduced their computational complexity by applying Winograd's minimal filtering algorithm [11, 12, 13, 14] or a Discrete Fourier Transform (DFT) [15]. While both of these techniques can reduce the number of multiplications required to implement a convolution, they limit our flexibility to perform other optimizations such as precision reductions, and make it more complicated to support a wide variety of convolution configurations (stride, kernel shape, etc.) [14]. In addition to the limits on flexibility, a recent trend in CNN architecture has been to separate convolutions into depthwise and and pointwise convolutions [16]. Of these two operations, pointwise convolutions are more computationally expensive, but their complexity cannot be reduced with these transforms. As a result, we look only at direct convolution methods.

A. Scatter or Gather Convolution

To perform a direct convolution with sparse weights we have the option to either (a) gather the correct activations for a group of weights and multiply and accumulate them in place as shown in Figure 1a, or (b) multiply all of the weights applicable to a particular input channel with the activations in that input channel and scatter them to a buffer for accumulation, like in [9] and as shown in Figure 1b. While the latter may be an efficient choice for an ASIC, on an FPGA we would like to make use of as many hardened resources as possible. The Stratix 10 DSP blocks include high precision accumulators as well as internal interconnections, called chain-out and chain-in, that allow efficient systolic accumulations within a DSP column, saving power and routing resources. If we were instead to scatter and accumulate into a buffer outside the DSP column we would require additional soft logic to perform the scatter and addition. Additionally, accumulation would require both a read and a write every cycle, and streaming completed data out would require another port. Such a 3-port RAM could be implemented with Stratix 10's quad port RAMs; however, each RAM would then have a maximum width of only 10 bits, which is not suitable for accumulation. For these reasons we elected to perform a gather-based convolution.


Dense Activations
Sparse Weights
for one output channel and multiple input channels
Gather to Match Weights and perform dot product
Multiply all weights with all activations in one input channel, add positions together to create addresses, and accumulate into another buffer
Sparse Weights
for multiple output channels and one input channel
One Input Channel of Activations

(a) Sparse Gather-based Convolution showing full computation of a single output.

(b) Sparse Scatter-based Convolution showing computation of many partial outputs.

Fig. 1. Comparison of Scatter and Gather-based Direct Sparse Convolutions.

B. Activation Partitioning

After selecting a convolution method we had to determine a data layout and partitioning scheme to process activations in parallel. We qualitatively evaluated three different partitioning schemes:

1) *Broadcast:* A scheme similar to Intel's DLA [12] that broadcasts activations to PEs from a global buffer and parallelizes multiply accumulate operations across output channels.
2) *Spatial Partition:* A scheme similar to SCNN [9] that partitions activations across PEs along the width and height dimensions for a single layer.
3) *Layer-Pipeline:* A scheme similar to that used by Zhang et al. [17] and Venieris and Bouganis [18] that partitions activations in the width and height dimensions across all layers simultaneously.

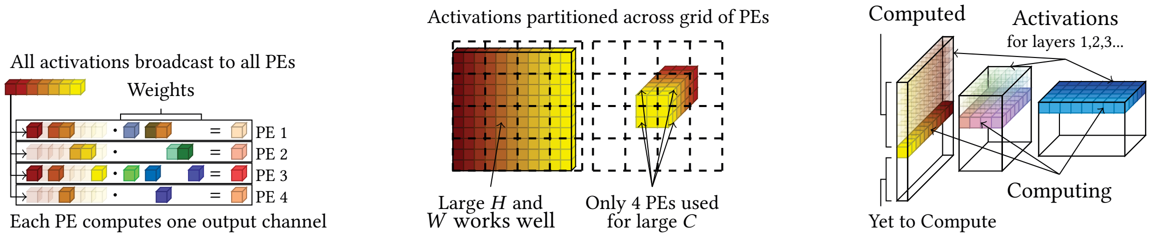
1) Broadcast: Intel's DLA [12] uses a PE architecture that streams and duplicates input features across multiple processing elements that each compute a different output channel. *Broadcast* works very well for a dense accelerator; however, with a sparse accelerator targeting around 85% weight sparsity, only 15% of the activations are used per output channel computation, as illustrated in Figure 2a. Since each PE would only use 15% of the activations broadcast to it, we would either need to have that PE compute multiple output channels serially (which constrains our ability to parallelize the computation) or we would need to increase the activation distribution bandwidth, at a substantial hardware cost, to match the throughput of the DSPs. Additionally, if we parallelize across output channels then we need each processing element to perform its own address calculations, which are more complicated and expensive for a sparse accelerator. From this assessment of the *Broadcast* architecture we conclude that a hardware efficient sparse accelerator needs to (a) minimize activation movement since activation re-use is relatively much lower in a sparse accelerator, and (b) share address computations for a large number of output activations computed in parallel.

2) Spatial Partition: SCNN [9] is a sparse ASIC accelerator that minimizes activation movement by partitioning input activations across tiled PEs in their height and width dimensions for a single layer. In this architecture activations needed by multiple PEs are directly sent to adjacent PEs. While this solves the activation bandwidth issue we had with *Broadcast*, this partitioning scheme has a PE under-utilization issue since the activations cannot be split across many PEs when the height and width dimensions of the activations shrink. Figure 2b shows two sets of activations being partitioned across a PE array. The one with large height and width dimensions works well, but the second one with many channels but small height and width dimensions only utilizes 4 PEs.

3) Layer-Pipeline: The last partitioning we considered was to build a fixed pipeline of layers and pass activations directly between the stages. Figure 2c shows how we can have multiple stages each computing a portion of different layers, which we call a partition, in parallel. Notice that the earlier layers will compute multiple partitions before later stages begin (since they are waiting for data from prior layers). The primary disadvantage of this architecture is that it requires more memory bandwidth for weights than other architectures. Each of the *Computing* (see Figure 2c) partitions require the entire set of weights to finish an output line. To reduce weight memory bandwidth requirements other accelerators typically load a set of weights and multiple input images, then use the weights to complete output activations for multiple inputs. By contrast, the *Layer-Pipeline* architecture uses all of the weights to complete only a portion of a single input. It then needs to load all of the weights again to complete the next portion of that input.

C. Comparison

The *Broadcast* architecture is extremely efficient for dense designs, but we could not determine a way to maintain its shape flexibility (input, output and weight tensor shapes), add support for zero-skipping, and not over-provision the hardware to the extent that we lose the physical design advantages of the architecture. The *Spatial Partition* addresses activation distribution issues, reducing one hardware inefficiency, but it yields low efficiency for layers with small spatial dimensions. It may be possible to add control logic to parallelize channels across the processing elements, but that increases the activation transfer distances and requires us to over-provision hardware. The *Layer-Pipeline* architecture addresses both of these issues by tailoring hardware specifically for each layer and directly transferring activations between the pipeline stages. The pipeline architecture does require more memory bandwidth for the weights than the other architectures, but we can mitigate


All activations broadcast to all PEs
Weights
PE 1
PE 2
PE 3
PE 4
Each PE computes one output channel
Activations partitioned across grid of PEs
Large H and W works well
Only 4 PEs used for large C
Computed
Activations for layers 1,2,3...
Computing
Yet to Compute

(a) *Broadcast* (b) *Spatial Partition* (c) *Layer-Pipeline*

Fig. 2. Overview of Activation Partitioning Architectures Explored.

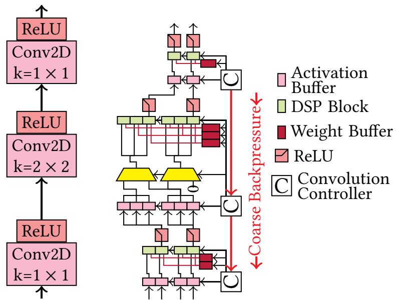


Fig. 3. Left: a simple example section of an input CNN. Right: the corresponding coarse-grained hardware pipeline. Each convolution circuit has its own buffer for data-reuse that also enables latency insensitive data transfers between the stages of the pipeline.

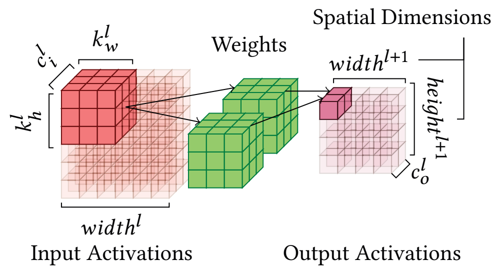


Fig. 4. A slice of an input activation and the corresponding output activation slice for a set of weights

this issue by storing all weights on-chip. Other accelerators, like Brainwave [19], have taken this approach, and simply use multiple chips if a network is too large to fit on one. Precision reductions and our support for zero-value weight compression further lower the cost of this approach. As a result, we elected to build an accelerator with a *Layer-Pipeline* architecture.

IV. Detailed Implementation

This section introduces the layer-pipelined HPIPE architecture, then provides an overview of the data organization and compute order choices made to facilitate zero-weight skipping with minimal overhead. HPIPE statically partitions device resources and instantiates custom hardware for each node in a CNN graph. These nodes can be convolutions, element-wise operations such as additions or multiplications, or nonlinear activation functions such as Rectified Linear Units (ReLU) or Swish ($\frac{x}{1+e^{-x}}$). The hardware units that implement these nodes directly transfer data to each other as described by the CNN graph, without writing to any global buffers. We arrange the data and order the computation to allow the hardware for many layers to simultaneously process a single input to the network, lowering processing latency.

A. Data Flow

Figure 3 shows 3 layers from an example CNN on the left, and a simplified diagram of the hardware that implements these layers on the right. The hardware is a coarse-grained pipeline with latency insensitive data transfers between each of the stages. The convolution layers must buffer data for re-use, and have limited buffer space, so their controllers send a coarse-grained backpressure signal to the hardware that produces its inputs. When deasserted, this backpressure signal allows a large non-contiguous burst of data to come from the prior layer, allowing for deep pipelining without any stall signals and clock enables.

Avoiding stall signals and clock enables allows our architecture to utilize the abundant but functionally limited interconnect (hyperflex) registers included on Stratix 10 devices. This allows our registers to be placed into a greater number of possible locations that can more flexibly balance timing paths. Omitting stall signals also generally boosts f_{max} since stall signals must typically fan-out to a large number of registers that span a large area.

Each stage in HPIPE processes one line of output data at a time. We call this an output channel group and it has a shape of $1 \times W \times C_o$, where W is the output width and C_o is the number of output channels. Additionally, all modules contain both a controller and a data path. The controllers typically compute addresses, load weights and biases, and communicate with their producers and consumers.

B. Layer Pipeline

To build a pipeline that overlaps the computation of many layers at once, the hardware for each layer must prioritize the completion of activations that allow later layers to do the same. The smallest slice of an activation that a compute unit needs in order to complete a slice of the output activation is a slice with the same dimensions as the dimensions of the weights (excluding c_o). Figure 4 shows a $6 \times 6 \times 3$ input activation for a convolution layer with weights of shape $3 \times 3 \times 3 \times 2$ along

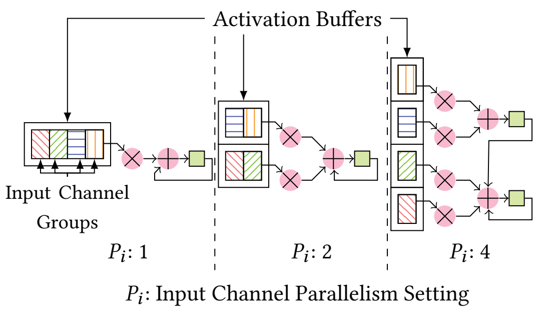


Fig. 5. Partial convolution compute unit with input channel parallelism settings ranging from 1–4

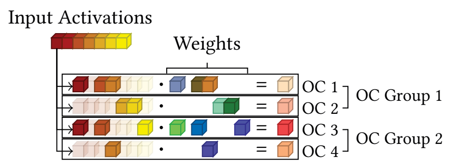


Fig. 6. Illustration of activations corresponding to weights in different output channels showing why increasing output channel parallelism requires duplication of entire activation buffers

with the output activation tensor for the layer ($4 \times 4 \times 2$). By prioritizing completion of slices of shape $k_h^{l+1} \times k_w^{l+1} \times c_o^l$—where l denotes the current layer, and $l+1$ is a consumer of its output—we can provide later layers with data they need to finish outputs earlier. This simultaneously reduces the memory buffering needed between layers and minimizes pipeline warm-up latency.

C. Parallelism

Our architecture leverages parallelism across c_i, c_o, and the width of the output activation, W. We fully parallelize the computation across the width of the output activation ($P_w = W$), but the parallelism across c_i and c_o are configurable by parallelism factors P_i and P_o. These parallelism factors control how many groups of input channels, $G_i^{p_i}$, and output channels, $G_o^{p_o}$, should be executed in parallel. The groups are sets of input or output channels that are combined to create $P_i * P_o$ slices, $G_t^{p_i,p_o}$, of the weight tensor, $\mathbf{w}$, as shown in Equation 1.

$$\begin{aligned} G_t^{p_i,p_o} &= \{\mathbf{w}_{h,w,i,o} \forall h \in K_h, w \in K_w, i \in G_i^{p_i}, o \in G_o^{p_o}\} \\ K_h &:= \{0,...,k_h-1\}, K_w := \{0,...,k_w-1\} \\ p_i &\in \{0,...,P_i-1\}, p_o \in \{0,...,P_o-1\} \end{aligned} \tag{1}$$

In addition to input and output channel parallelism, we fully parallelize the computation across the width of the output activation. This maximizes address computation efficiency and our ratio of computation to activation buffering. We then rely only on P_i and P_o to increase and balance throughput among all the layers. This is similar to Zhang et al. [17], but our cost function for optimizing the throughput of each layer is different due to our support for sparsity.

Figure 5 shows how we can divide input channels into groups and put each group in a different buffer to increase parallelism. Each of these buffers requires a different read address, but the depth of the buffer decreases as we increase the parallelism setting, meaning the total buffer size remains the same. By contrast, parallelizing across output channels requires duplication of the buffers.

To see this, consider Figure 6, which shows a sparse kernel with 7 weights (some of which are 0) per Output Channel (OC). If we group OC 1 and OC 2 together (for one compute unit to calculate serially), and group OC 3 and OC 4 together for another compute unit, we will need 5 (out of 7) inputs for the first group, and 4 (out of 7) for the second, with group sizes of only 2. In practice we may have between 32-2048 output channels, with P_o settings of 1-80. If we allocated all of our DSPs to parallelizing across output channels we would have output channel group sizes of around 10-100. This means we typically need all of the input activations at some point while processing each output channel group even in the case where we target a large device and parallelize the output channel computation as much as possible. Consequently, activation buffer duplication is necessary when we parallelize across output channels.

This is different than it would be for an accelerator that does not skip zero weights since such an accelerator can load a single activation and fan it out to be processsed with different weights for all output channels in parallel. Nevertheless, parallelizing across output channels is a useful knob to have to further increase parallelism and to alleviate routing congestion issues in networks where there are spare on-chip RAMs.

D. Physical Design Parameters

To achieve high throughput, we need not only to fit many compute units and their associated buffers on chip, but also achieve a high clock frequency. Most delay in an FPGA is due to the interconnect, so it is essential that our compiler predict and optimize interconnect fanout and span (width and height covered by the net) as it chooses parallelism, interconnect multiplexing, and pipelining parameters. The FPGA we target, Stratix 10, also has abundant, but functionally limited, registers within the interconnect itself, making interconnect-aware pipelining a powerful optimization that we wish to exploit.

Our architecture supports arbitrary pipelining requiring no stall or reset logic (allowing the registers to be placed into the Stratix 10 interconnect registers) not only between modules, but within compute blocks as well. Figure 7 shows one configuration of our convolution module where $P_i = 4$, $P_o = 1$, and $k_w = 3$. There are eight sets of signals within the module that can be arbitrarily pipelined, but since the latencies along each of these paths must align with the latencies along the other paths their cycle delays become functions of their own pipelining requirements and the requirements of other signals in the module. Our compiler can estimate the cycle delay required to minimize fanout and span for each of the eight paths, then compute cycle delays greater than or equal to these values to ensure functional correctness. This gives our compiler enough flexibility to maintain a high F_{max} in all modules even as the parallelism settings, buffer depths, stride, and weight and output shapes change.

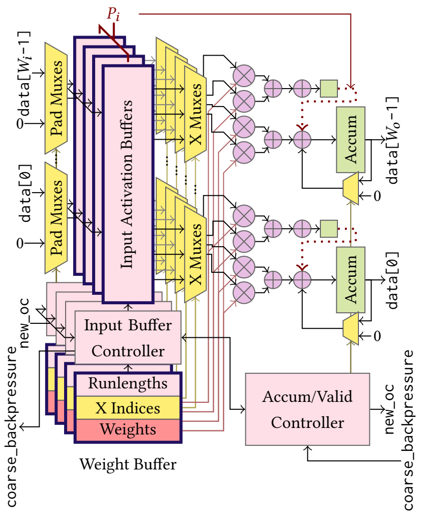


Fig. 7. One configuration of our convolution module

Our compiler can also determine the structures of the circuits that distribute fanout across multiple pipeline stages. It can select between a tree structure and a serial distribution structure. The serial structure is more efficient (uses fewer registers to distribute data across large distances), but the signal gets distributed to the sinks in different cycles. For write data arriving at the input activation buffers our hardware can tolerate these asynchronous arrivals. For the weight fanout we must realign the data with shift registers implemented in LUTRAMs (20×32 RAMs made out of the LUTs that make up most of the area on an FPGA). This increases the cost of the serial fanout structure, but it can still be more efficient than the tree structure when the number of sinks on the net is low and the the span is large. Our compiler must determine which structure to use in these situations.

V. Network Compiler and Tool Flow

Figure 8 shows a high level overview of the inputs to, and stages and outputs of our compiler. It takes in a TensorFlow graph along with configuration information for the target device and an optional file to specify the desired fixed-point numerical format for each layer (integer and fractional bits may be specified). The compiler then optimizes the TensorFlow graph for inference by merging some nodes and mapping the nodes to HPIPE modules. The device allocator then analyzes the parameters and determines the parallelism settings for each layer to balance throughput. Using these parameters it then builds an in-memory RTL graph representation of the coarse-grained pipeline that implements the CNN. We then optimize the graph by converting the user-specified RAMs into smaller RAMs that map directly into the physical RAMs on the FPGA. We find that some of our large RAMs can limit frequency if we rely on the Quartus mapping since Quartus does not retime registers on the read path into the read muxing for deep RAMs. We also duplicate registers in pipeline stages leading to high fanout nets to distribute the loading and the span across a

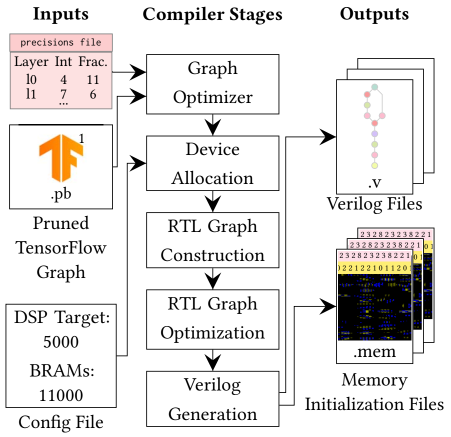


Fig. 8. Compiler inputs, outputs, and stages

greater number of timing endpoints. After these optimizations, our compiler converts the in-memory graph representation to a Verilog description and generates corresponding memory initialization files, timing constraints, and Quartus settings. We do not have space to detail all of the components of the compiler, so the following section will detail the Device Allocation stage that selects parameters to create a pipeline where the throughput of each of the stages is approximately equal.

A. Device Allocation

Figure 9 shows cycle latencies from independent simulations of each of the layers in our 85% sparse ResNet-50 model with the default parallelism settings ($P_i = 2, P_o = 1, P_w = W$). In the Device Allocation stage, we attempt to balance these latencies by increasing the parallelism settings for the highest latency layers. Figure 9 also shows the latencies after we have balanced them, and the corresponding device resource utilization for each layer. Like Zhang et al. [17] we achieve this by modelling the latency of each of the layers, sorting them by their latency, and repeatedly increasing the paralellism settings for the highest latency node until we achieve the target resource count; however, to efficiently skip zero-weights our hardware imposes certain constraints that mean we do not always scale performance linearly with additional hardware. To accurately balance the pipeline we need accurate models of this nonlinear performance scaling, and we need to account for the sparsity pattern in the actual weights for each layer.

In the case where latency scales linearly with parallelism factors and there is no zero weight skipping, the latency of a convolution layer, l, follows Equation 3. Where W and H are output width and height, and $\mathbf{w}$ is the weight tensor. We define a parallelism factor P composed from input channel, output channel, and width parallelism factors as shown in Equation 2. We can adapt Equation 3 to handle 0 skipping by counting only the nonzero weights in the weight tensor as shown in Equation 4; however, this does not account for the fact that hardware generally cannot perfectly exploit sparsity. To account for this, we further adapt the equation as shown in Equation 5. In this case we divide our input channels into P_i

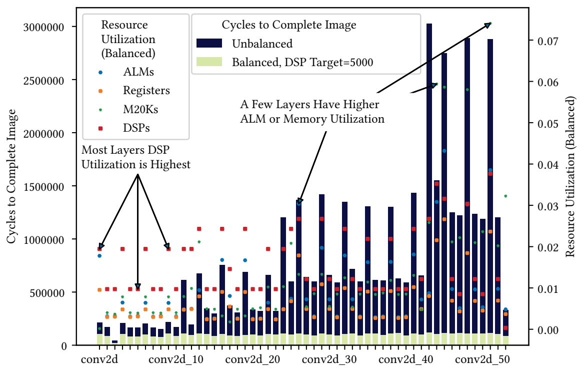


Fig. 9. Comparison of Individual Layer Latency Before and After Balancing, and Resource Utilization per Layer as Percentages of the Total Chip Resources

input channel groups, $G_i^{p_i}$, and put these groups into a set O. Our hardware can process the groups in parallel, but all of the results from each group within a single output channel must be summed together. Since our hardware uses the dedicated systolic accumulation chain in the DSP columns of Stratix 10 devices to accumulate values, we cannot begin to process the weights for the next output channel until all groups have finished processing. To model this, we sum the maximum latencies over the set of input channel groups, O, within each output channel and sum these across all output channels to represent the latency for each point in the spatial dimensions. We have removed $W * P_w^{-1}$ since $W = P_w$ in our hardware, so the spatial component of delay reduces to H. Finally, we divide the latency by the output channel parallelism setting P_o. While this final step does not accurately model the latency with $P_o \gtrapprox \frac{c_o}{4}$, we typically find $P_o << \frac{c_o}{4}$ for even small CNNs on large FPGAs.

$$P = (P_i * P_o * P_w) \tag{2}$$

$$l_d = P^{-1} * W * H * \sum_{w \in \mathbf{w}} 1 \tag{3}$$

$$l_s = P^{-1} * W * H * \sum_{w \in \mathbf{w}} \begin{cases} 0 & w = 0 \\ 1 & else \end{cases} \tag{4}$$

$$l_g = P_o^{-1} * H * \sum_{o=0}^{c_o - 1} \max_{G_i \in O} \sum_{i \in G_i} \sum_{h=0}^{k_h - 1} \sum_{w=0}^{k_w - 1} \begin{cases} 0 & \mathbf{w}_{h,w,i,o} = 0 \\ 1 & else \end{cases} \tag{5}$$

VI. Performance Results

We compare to six other accelerators, with results on three different CNNs. First, we evaluate our throughput on Resnet-50, a popular and high accuracy but compute-intensive CNN. The highest performance GPU and FPGA implementations of Resnet-50 are dense, so we compare to these, but our implementation leverages sparsity and achieves higher throughput at a modest accuracy cost. Second, we compare to a prior sparse-CNN FPGA accelerator and show that we can exploit all the device DSPs, while their higher logic utilization limits the number of DSPs they can use. Third, we compare throughput vs. a GPU and a prior FPGA accelerator on the compute-efficient and *dense* MobileNet CNNs (V1 vs. a GPU and V2 vs. the prior FPGA accelerator). This comparison shows we outperform these accelerators even without leveraging sparsity and while running at 2x the precision.

Table I shows a breakdown of resources, utilization, network information, and performance for all of the accelerators and networks we compare to. The following sections discuss the results of the table, broken down into the most relevant comparisons.

A. Highest Throughput Accelerators

We outperform all prior FPGA accelerators on ResNet-50 by a wide margin of 8-45x; in some cases we use a larger device, but even with scaling throughput linearly with device size (which is likely overly optimistic) we outperform them by 1.5-11x. The closest competitor is the V100, the highest performance shipping GPU, which most FPGA prior work has not compared to. We outperform them by nearly 4x for throughput at a batch size of 1, and match their throughput when they use a batch size of 8 (at which point our latency is approximately half of theirs). Our performance comes from a combination of our high operating frequency, high DSP utilization, and our ability to exploit network sparsity. The fact that we have the highest operating frequencies (430 to 580MHz) despite automatically generating customized hardware speaks to the utility of our physically-aware automatic pipelining generator tool.

Table I shows that we achieve throughput at a batch size of 1 that is nearly 4x that of the V100 GPU despite it having more than 10x the peak TOPs. We also compute the effective TOPs as the throughput in images per second times the number of operations in the CNN (CNN GOPs), which shows that they are only able to leverage about 7% of their total compute at a batch size of 1. At a batch size of 8 they have roughly the same throughput as HPIPE, but at roughly 2x the latency.

The Brainwave [19] accelerator achieves high performance by packing many multipliers into the lookup tables on the FPGA. Our throughput is about 8x theirs. While they use an older and smaller device, even perfect scaling would yield around 2,700 images per second*

DLA-Like [21] leverages the Winograd transform to reduce the number of multiplications required to perform convolutions, but even with this optimization the effective TOPs is roughly half the peak TOPs.

By contrast, DNNBuilder [17], which also uses a layer-pipeline approach manages to fully utilize their 4 Peak TOPs of performance even though VGG-16 is a more memory bound CNN architecture (since it has three large fully-connected layers). Still, this effective TOPs number is about 8x lower than ours. This is partially due to the zero-skipping support in our

*We arrive at 2,700 images per second by using the 4.9x peak TFLOPs increase they report for moving a language model accelerator from Arria 10 to Stratix 10. This is likely to be very optimistic since utilization typically decreases as parallelism increases.

TABLE I
COMPARISON TO OTHER HIGH PERFORMANCE CNN ACCELERATORS.

	NVIDIA [20]	NVIDIA [20]	Brainwave [19]	DLA-Like [21]	Wu et al. [22]	DNNBuilder [17]	Lu et al. [1]	**HPIPE (Ours)**	**HPIPE (Ours)**	**HPIPE (Ours)**
Device	V100	V100	A10 1150	A10 1150	ZU9	KU115	ZU9	S10 2800	S10 2800	S10 2800
Technology	TSMC 12N	TSMC 12N	TSMC 20nm	TSMC 20nm	20nm	20nm	20nm	Intel 14nm	Intel 14nm	Intel 14nm
Vendor	NVIDIA	NVIDA	Intel	Intel	Xilinx	Xilinx	Xilinx	Intel	Intel	Intel
Architecture	Volta	Volta	Arria 10	Arria 10	Ultrascale+	Ultrascale+	Ultrascale+	Stratix 10	Stratix 10	Stratix 10
Device BRAM (Mb)	–	–	53	53	32	76	32	229	229	229
Device DSPs	–	–	1,518	1,518	2,520	5,520	2,520	5,780	5,780	5,780
Logic Utilization	–	–	–	40%	59%	39%	92%	63%	40%	38%
BRAM Utilization	–	–	–	71%	40%	81%	48%	96%	37%	40%
Used DSPs	–	–	1,518 (100%)	1,376 (91%)	2,070 (82%)	4,318 (78%)	1,144 (45%)	5,022 (87%)	5,133 (89%)	2,964 (51%)
Frequency (MHz)	1,246	1,246	–	345	333	235	200	580	430	530
Network	Resnet-50	MobileNet-V1	Resnet-50	Resnet-50	MobileNet-V2	VGG-16	Resnet-50	Resnet-50	MobileNet-V1	MobileNet-V2
Sparsity	0%	0%	0%	0%	0%	0%	76.5%	85%	0%	0%
Winograd	No	No	No	Yes	No	No	No	No	No	No
Precision	8-Bit	8-Bit	11-Bit	16-Bit	8-Bit	8-Bit	16-Bit	16-Bit	16-Bit	16-Bit
Format	Fixed	Fixed	Block Float	Fixed	Fixed	Fixed	Fixed	Fixed	Fixed	Fixed
Throughput* (Peak TOPs)	125	125	–†	1.9‡	2.8‡	4.0	0.4	11.7	8.7	6.2
Throughput (B=1, im/s)	1156	4,605	559	119	810	130	–	4551	5,157	6,023
Latency ($B = 1$, ms)	0.87	0.22	1.8	10.5	–	–	–	1.08	0.65	0.86
CNN GOPs	7.3	1.1	7.3	7.3	0.7	30.7	7.3	7.3	1.1	0.7
Effective§ TOPs	8.4	5.1	4.1	0.9	0.6	4.0	–	6.8/33.2	2.1/10.6	0.8/3.9
Top-1 Accuracy	74.93%	–	$\sim$ 76%¶	–	68.1%	69.2%	$\sim$ 76%¶	71.9%	72.0%	72.0%

* Peak TOPs here is the number of trillions of multiplications and additions performed by the multipliers and adders used to perform the core computations in the CNN.
† Fowers et al. [19] does not provide peak TOPs for their Resnet-50 accelerator, and we cannot estimate it from their DSP usage since they use a large but unspecified number of multipliers built in the soft logic.
‡ Boutros et al. [21] and Wu et al. [22] do not provide peak TOPs, so we estimated their numbers from their frequency, DSPs used, and multipliers packed into each DSP.
§ We compute Effective TOPs as the Throughput (B=1, im/s) times the CNN GOPs. For designs exploiting pipeline parallelism we also provide a single image effective TOPs number that is computed as the inverse of the single image latency multiplied by the CNN GOPs. Zhang et al. [17] and Wu et al. [22] also exploit pipeline parallelism, but they do not provide latency numbers, so we could not compute the single image TOPs for them.
¶ Fowers et al. [19] and Lu et al. [1] do not state accuracy, but state degradation is negligible.

accelerator, but also partially due to our higher DSP utilization and higher frequency. We believe our higher frequency is a result of the physical awareness of our compiler. While they have scripts that select parameters for hand-written Verilog templates, these parameters do not control construction of custom fanout structures like our compiler does.

In addition to the differences between VGG-16 and ResNet-50, there are slight differences in the ResNet-50 models used. We collected our accuracy by running the 50,000 image ImageNet validation set on actual hardware, using PCIe to transfer data to and from the accelerator. Our top-1 and top-5 accuracies of 71.9% and 90.8% match the input TensorFlow model when it is run on a GPU. While this is lower than the typical ResNet-50 accuracy, Zeng and Urtasun [23] report that they are able to train 85% sparse ResNet-50 models that have no accuracy loss.

B. Sparse CNN on FPGA

Lu et al. [1] is an important comparison because it is the only other work we are aware of that implemented a sparse CNN accelerator on an FPGA that can support generic CNN architectures. That being said, they do not provide latency or images per second throughput numbers for their accelerator, so we mostly have to compare resource utilization and peak TOPs. Their high logic utilization limits them to only 45% of the DSPs, and their frequency is less than half of ours. While they use a smaller device than us, they have a chart that shows logic and DSP utilization remain roughly constant as they scale their design to use more processing elements. We expect that we achieve roughly 6x the throughput per area based on the nearly 3x higher frequency and nearly 2x DSP utilization.

C. Dense MobileNet-V1 and MobileNet-V2

On MobileNet-V1 we compare again to the V100 GPU. On this model we do not prune any weights and we find we can maintain the full accuracy of the input network. NVIDIA does not report accuracy, and they quantize to 8-bits, so it may be lower. For latency we are behind by 0.43ms (due to our pipeline warm-up latency), but we demonstrate higher throughput despite running at 2x the precision and not leveraging the sparse acceleration capabilities of our accelerator.

On MobileNet-V2 we compare to Wu et al. [22]. We run at twice their precision and maintain the full accuracy of the input network, while they lose 3.9% top-1 accuracy due to their 8-bit quantization. We achieve 7.4x the throughput while using only 43% more DSP blocks and again not leveraging the zero-weight skipping capabilities of our accelerator. We believe this performance derives from a combination of a) our higher frequency of 530MHz vs. 333MHz and b) under-utilization of their DSP blocks due to imperfect mapping of some layers onto their PEs.

Unlike MobileNet-V1 and Resnet-50, for MobileNet-V2 we were unable to achieve 87-89% DSP utilization on the S10 2800. We did not run out of BRAMs or logic, instead the low DSP count is due to a limitation of the current version of our hardware generator. After parallelizing to use nearly 3,000 DSPs the node with the lowest throughput in the network becomes a depthwise convolution to which we cannot currently exploit any additional parallelism. Even with this limitation we find our CNN mapping efficiency and high frequency make our accelerator outperform prior work by a wide margin.

VII. Conclusion

In this paper we have demonstrated that a gather-based layer-pipelined approach to CNN acceleration can leverage sparsity without the added hardware cost from prior works. Together with our high operating frequencies—which are a result of our physically aware CNN hardware generator—this approach enables inference throughput at a batch size of 1 that is 4x higher than the fastest GPU for machine learning on a large but sparse CNN. We also outperform all prior FPGA accelerators to which we can compare, even when assuming optimistic linear scaling of their throughput with device DSP count. On smaller and more efficient dense models that do not leverage our 0-weight skipping we still achieve higher throughput than the GPU and another FPGA accelerator while running at twice the precision. Our variable precision support and throughput balancing algorithms will allow future accelerators based on this architecture to prune weights only from layers where accuracy does not suffer, and reduce the precision in particular layers where higher precision is less important. Looking towards future FPGA architectures with DSP support for lower precision multipliers, these features could provide further performance improvements per area of 2x or more.

VIII. Acknowledgements

We would like to thank the National Sciences and Engineering Research Council of Canada (NSERC) and Intel Industrial Research Chair in Programmable Silicon for funding support.

References

[1] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, "An efficient hardware accelerator for sparse convolutional neural networks on FPGAs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2019, pp. 17–25.

[2] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," in *ICLR*, 2016.

[3] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 243–254. [Online]. Available: https://doi.org/10.1109/ISCA.2016.30

[4] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 75–84. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021745

[5] M. Zhang, L. Li, H. Wang, Y. Liu, H. Qin, and W. Zhao, "Optimized compression for implementing convolutional neural networks on FPGA," *Electronics*, vol. 8, p. 295, 03 2019.

[6] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: ACM, 2019, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/3289602.3293898

[7] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 821–834. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304028

[8] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 20:1–20:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3195638.3195662

[9] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 27–40. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080254

[10] A. Aimar, H. M. Z. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I.-A. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S.-C. Liu, and T. Delbruck, "Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, pp. 644–656, 2017.

[11] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp.

4013–4021.

[12] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl^{TM}deep learning accelerator on arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 55–64. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021738

[13] L. Lu and Y. Liang, "SpWA: An efficient sparse winograd convolutional neural networks accelerator on FPGAs," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: ACM, 2018, pp. 135:1–135:6. [Online]. Available: http://doi.acm.org/10.1145/3195970.3196120

[14] K. Vincent, K. Stephano, M. A. Frumkin, B. Ginsburg, and J. Demouth, "On improving the numerical stability of winograd convolutions," in *ICLR*, 2017.

[15] C. Zhang and V. Prasanna, "Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 35–44. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021727

[16] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. Long Beach, California, USA: PMLR, 09–15 Jun 2019, pp. 6105–6114. [Online]. Available: http://proceedings.mlr.press/v97/tan19a.html

[17] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3240765.3240801

[18] S. I. Venieris and C. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2016, pp. 40–47. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/FCCM.2016.22

[19] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 1–14.

[20] NVIDIA Corporation. (2019) Nvidia tesla deep learning product performance. [Online]. Available: https://web.archive.org/web/20190817114937/https://developer.nvidia.com/deep-learning-performance-training-inference

[21] 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*, vol. 11, pp. 1–23, 12 2018.

[22] D. Wu, Y. Zhang, X. Jia, L. Tian, T. Li, L. Sui, D. Xie, and Y. Shan, "A high-performance cnn processor based on FPGA for mobilenets," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2019, pp. 136–143.

[23] W. Zeng and R. Urtasun, "MLPrune: Multi-layer pruning for automated neural network compression," 2019. [Online]. Available: https://openreview.net/forum?id=r1g5b2RcKm