HaShiFlex: A High-Throughput Hardened Shifter DNN Accelerator with Fine-Tuning Flexibility

JONATHAN HERBST, Brown University
MICHAEL PELLAUER, NVIDIA
SHERIEF REDA, Brown University

We introduce a high-throughput neural network accelerator that embeds most network layers directly in hardware, minimizing data transfer and memory usage while preserving a degree of flexibility via a small neural processing unit for the final classification layer. By leveraging power-of-two (Po2) quantization for weights, we replace multiplications with simple rewiring, effectively reducing each convolution to a series of additions. This streamlined approach offers high-throughput, energy-efficient processing, making it highly suitable for applications where model parameters remain stable, such as continuous sensing tasks at the edge or large-scale data center deployments. Furthermore, by including a strategically chosen reprogrammable final layer, our design achieves high throughput without sacrificing fine-tuning capabilities.

We implement this accelerator in a 7nm ASIC flow using MobileNetV2 as a baseline and report throughput, area, accuracy, and sensitivity to quantization and pruning—demonstrating both the advantages and potential trade-offs of the proposed architecture. We find that for MobileNetV2, we can improve inference throughput by 20× over fully programmable GPUs, processing 1.21 million images per second through a full forward pass while retaining fine-tuning flexibility. If absolutely no post-deployment fine tuning is required, this advantage increases to 67× at 4 million images per second.

Additional Key Words and Phrases: Deep Learning Accelerators, Efficiency, Quantization, Fine Tuning

1 Introduction

Hardware accelerators have become a cornerstone of modern deep learning, largely because of the computational intensity associated with training and inference in neural networks. As datasets continue to grow and model architectures become more complex, general-purpose processors such as CPUs often struggle to deliver the high throughput and energy efficiency necessary for real-time or large-scale applications. By contrast, specialized hardware designs can dramatically reduce the time and energy required to execute neural network operations.

Specialized architectures built for neural network processing come in several broad categories. First, there are GPU-based solutions, which combine Turing-complete streaming multiprocessors with integrated tensor cores to accelerate the matrix operations commonly found in neural network workloads. Second, there are FPGA-based accelerators that gain efficiency through reconfigurable logic, allowing researchers to tailor the hardware pipeline to specific neural architectures. Third, specialized ASICs like Google's TPU and various neural processing units are designed from the ground up to address specific patterns of computations in neural networks. Each approach offers distinct trade-offs in terms of flexibility, power efficiency, and raw computational capabilities. Each of these solutions store the neural network weight parameters in dynamically writable RAM memories that are loaded with the weight values from the most recent version of the model at runtime. Multi-layer neural networks are time-multiplexed through the accelerator hardware using the best-known scheduling policy, updating the weight memories with the relevant parameters for each layer.

Authors' Contact Information: Jonathan Herbst, Brown University, jonathan_herbst@brown.edu, c@example.com; Michael Pellauer, NVIDIA, mpellauer@nvidia.com; Sherief Reda, Brown University, sherief_reda@brown.edu.

2025. Manuscript submitted to ACM

In contrast, hardwiring the accelerator by fixing weights directly into the hardware can substantially simplify the underlying architecture. A fixed-weight design can eliminate all weight related memory transfer overheads by embedding learned parameters in logic. Furthermore, dedicating hardware resources to only those operations necessary for the fixed model can streamline data pathways via design-time constant propagation, further increasing performance. To accomplish this, different layers of the network are no longer time-multiplexed into the same datapaths and RAMs, but instead are spatially "unrolled" into different physical areas of the chip. Thus, while per-layer area efficiency increases, the overall area required to provision an entire DNN can be quite large. On the other hand, this spatial distribution of resources also results in a high-throughput pipelined execution across layers, and so the tradeoff can be favorable.

Hardwired accelerators offer compelling advantages for scenarios requiring high-performance, energy-efficient inference with relatively static or infrequently updated models. They are particularly well-suited for applications where the target model is stable for extended periods and speed or power efficiency is paramount—such as embedded systems performing continuous sensing or computer vision tasks, edge devices with stringent power budgets, or large-scale data centers running fixed workloads at massive scale. This combination of efficiency, reliability, and performance can enable real-time responsiveness and broader deployment possibilities, especially in domains where frequent retraining is not necessary or where hardware upgrade cycles naturally accommodate incremental updates to the accelerator design. Of course, the natural downside of this is that post-deployment hardware adaptability decreases, which can be a large problem in a fast-moving field like Deep Learning.

In this paper, we propose a high-throughput accelerator architecture that simultaneously addresses the downsides of overall area efficiency and hardware adaptability when using hardwired weights. To retain a degree of post design-time flexibility, we include a small neural processing unit (NPU) dedicated to handling the final classification layer, thus enabling on-the-fly adjustment of model outputs (i.e., fine-tuning). Furthermore, we leverage power-of-two quantization for hardwired weights, allowing us to dispense with multipliers or shifters entirely, as each shift operation becomes a simple hardware re-wiring. This dramatically increases per-layer area efficiency, thus making our design *the first to make full spatial unrolling of a realistic neural network feasible in a limited area budget.* It is our hope that our combination of hardening and flexibility will inspire future research in this rich design space. We summarize the specific contributions of this paper as follows:

- We present a novel hardwired CNN accelerator design that eliminates the need for storing and transferring weights. By quantizing weights to powers of two, all multiplication operations are replaced with rewiring, thereby reducing convolutions to a sequence of additions. We also show how batch normalization and non-linear activations can be seamlessly integrated into this framework.
- To preserve flexibility, we incorporate a small neural processing unit (NPU) that handles the final classification layer in a traditional manner. This approach offers adaptability in the output stage, mitigating some of the rigidity introduced by hardwiring the earlier layers.
- We analyze the area implications of our proposed architecture and identify widely-used CNN models that fit within reticle constraints, demonstrating how they can be effectively deployed to leverage the benefits of hardwired processing.

- Using MobileNetV2 as a baseline and implementing our design in a 7 nm ASIC flow, we provide a comprehensive set of experimental results evaluating throughput, area, accuracy, and flexibility. We also conduct a sensitivity analysis to assess how bit quantization and pruning affect these performance metrics.

The remainder of this paper is organized as follows. In the Background section, we review key concepts and existing accelerator architectures relevant to our proposed design. Next, in Section 3 we detail our hardwired architecture, including the quantization strategy and the integration of a small neural processing unit for added flexibility. In Section 4 we discuss training. In Section 5, we present comprehensive performance benchmarks, including throughput, area, accuracy, and sensitivity analyses. We then discuss our work within the broader research landscape in the Related Work section, discussing key differences and similarities to prior accelerator designs. Finally, Section 7, summarizes our main findings and suggests possible directions for future research.

2 Background

2.1 Convolutional Neural Networks

One of the most important types of Deep Neural Networks (DNNs) is the Convolutional Neural Network (CNN), formalized by Yann LeCun as a way to extract spatial information from images [17]. CNNs "slide" a filter over the previous layer's features, using several filters to separate information into planar "channels."

Throughout this paper, we use the following convention, defined in Eyeriss [6]: P and Q represent the convolutional output's height and width, M represents output channels, R and S represent the kernel's height and width, and H, W, and C represent the convolutional input's height, width, and number of channels respectively. In addition, we use O or $O^{P\times Q\times M}$ to represent the convolution output, W or $W^{M\times R\times S\times C}$ to represent the filter matrix, and I or $I^{H\times W\times C}$ to represent the convolution input. Thus we can write each output element in tensor algebra summation notation:

$$O_{pqm} = \sum_{r=0}^{R}\sum_{s=0}^{S}\sum_{c=0}^{C} W_{m,r,s,c} \cdot I_{p+r,q+s,c}$$

In practice, many accelerators map this operation onto a 2-dimensional matrix multiplication, known as the Toeplitz representation. We flatten the M filter weight matrices $W^{R\times S\times C}$ into the 2D matrix $W^{M\times RSC}$, and unroll the input image PQ times to get $X^{RSC\times PQ}$. Thus, another equivalent way to write the equation would be

$$O^{PQ\times M} = W^{M\times RSC} X^{RSC\times PQ}$$

after reshaping the results to a 3-dimensional format.

In order to calculate an output element, we need RSC filter weights and inputs, which must be loaded from memory. Because hardware devices only have limited on-chip storage, these values must be continuously streamed from main memory, which provides a memory bottleneck that reduces computational intensity.

2.2 Hardware-Friendly Quantization & Pruning

In order to reduce these memory costs (as well as increase area- and energy-efficiency) we can compress the model size through two techniques known as *quantization* and *pruning*. Quantization reduces the number of bits used to represent each weight, which sometimes also applies to the activations; this can also involve changing the representation type, for instance from floating-point to fixed-point. This applies across all weights in the model, although some layers can have different quantization than other layers, i.e., mixed-precision [4, 31, 33].

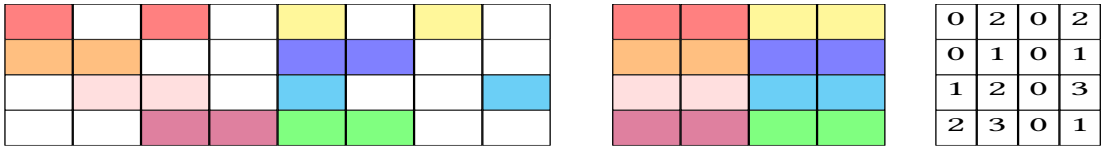


Fig. 1. 2:4 Weight Compression. We slice the matrix along its row axis in groups of four elements, then extract the two nonzero elements into a smaller matrix. Our metadata matrix contains two-bit indices of the nonzero values in their original groups.

Table 1. Qualitative comparison to existing DNN accelerators. See Figure 4 for detailed quantification of area unrolling costs.

Design Category	Design	Weight Value Strategy		Datapath Strategy	Multi-Layer Strategy	Throughput†	Qualitative Area Cost	
		Design-Time	Run-Time				Per Layer	Unrolled
PE Array	Eyeriss [6]	-	Dynamic	Multipliers (Row Stationary)	Time-Multiplexing	Medium	Full Die	**Infeasible**
	DianNao [3]	-	Dynamic	Multipliers (Weight Stationary)	Time Multiplexing	Medium	Full Die	**Infeasible**
	TPU [15]	-	Dynamic	Multipliers (Systolic Array)	Time Multiplexing	Medium	Full Die	**Infeasible**
Hardened PE Array	*Nonexistent*	Fixed	-	Constant-Propagated Multipliers	Space Unrolling	High	Medium	**Infeasible**
Shifter-Based	ShiftCNN [9]	-	Dynamic Po2	Shift-and-Add (ShiftALU)	Time Multiplexing	Medium	Medium	**Infeasible**
	Universal Shift [30]	-	Dynamic Po2	Shift-and-Add (conv2d-shift)	Time Multiplexing	Medium	Medium	**Infeasible**
	Jumping Shift [12]	-	Dynamic Po2	Shift-and-Add (2-bit barrel)	Time Multiplexing	Medium	Medium	**Infeasible**
This Work	*HaShiFix*	Fixed Po2	-	Rewire (Constant-Propagated Shift)	Space Unrolling	Highest	Very Low	Large
	HaShiFlex	Fixed Po2	Fine Tuning	Rewire (Constant-Propagated Shift) and Multipliers	Space Unrolling and Time Multiplexing	High	Low	Full Die

† Refers to throughput of completing a forward pass through all DNN layers.

The other major compression technique is pruning, which eliminates some proportion of the model's weights entirely. We can perform either *structured* or *unstructured* pruning, where the former involves clamping weights in some organized pattern and the latter allows us to clamp any of the weights without regard to organization. Both pruning formats reduce the number of multiplications needed, since we can ignore the zero weights, but they have limited benefits in many accelerators. In particular, GPUs compress in a 2:4 weight compression strategy, which preserves two weights out of every four elements in a row group. This means transferring half the weights and inputs instead of the full amount, but also requires us to transfer a compression matrix with the indices of each nonzero element (Figure 1). This means that for matrices $W^{PQ\times RSC}$ and $X^{RSC\times M}$, instead of transferring PQRSC + RSCM weights we transfer $PQ\frac{RSC}{2} + \frac{RSC}{2}M$ weights and $PQ\frac{RSC}{2}$ 2-bit indices in our metadata matrix.

In terms of compute cycles, 2:4 weight compression allows us to run a smaller matrix multiplication: instead of $W^{M\times RSC}X^{RSC\times PQ}$ we only compute $W^{M\times \frac{RSC}{2}}X^{\frac{RSC}{2}\times PQ}$, where the inner dimension is only halfsize. We model this cycle count with SCALE-Sim [28] in our results section to understand sparsity's effect on accelerator throughput.

2.3 Contrast to Existing DNN Accelerators

DNN accelerators fall into three subtypes: fully programmable (i.e., GPUs), fixed function multiplier arrays (i.e„ TPUs), and Field-Programmable or Coarse-Grained Reconfigurable Arrays (also called NPUs). Each of them has a main characteristic in common, which is a loadable array of processing elements, or PEs. To execute a complete forward pass through a multi-layer Deep Neural Network, these PEs load the parameters of each layer in sequence (i.e., time multiplexing of the hardware datapaths), passing output features from one layer into the next via on-chip storage.

Table 1 shows a detailed breakdown of why these PE approaches cannot be used directly to achieve our goals. Namely, while they achieve maximum post-deployment ability to change DNN parameters, overall forward pass throughput suffers because of this time multiplexing. Thus, when field updating of weights is not required, it is natural to investigate whether using design-time constant weight values and constant-propagation is feasible to increase throughput. This

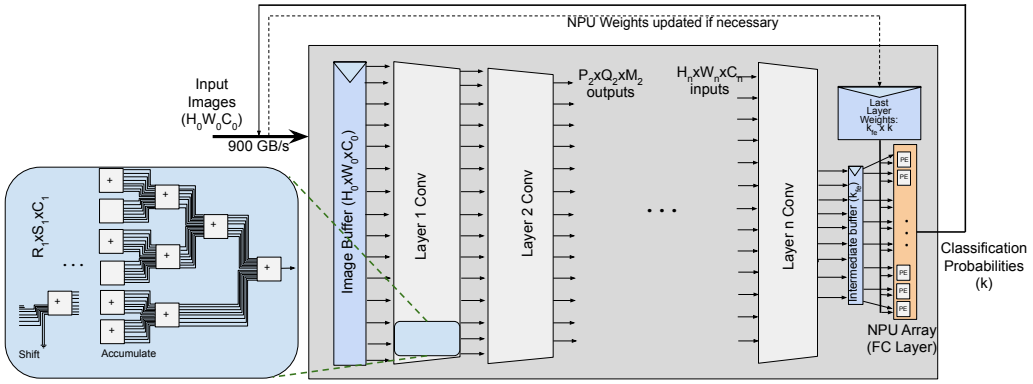


Fig. 2. An overview of the HaShiFlex ASIC design. Image inputs are loaded into on-chip memory, run through the hardcoded convolution layers, and the feature extractor outputs are stored in an intermediate buffer. These activations are run through a flexible NPU array that multiplies them against stored on-chip weights to compute the final classification probabilities, which are streamed off-chip.

approach requires spatially "unrolling" the hardware for each layer – as each layer has different fixed weight values. Unfortunately, multipliers themselves are large and area inefficient. This style of constant propagation does not result in sufficient area savings to allow for fitting an entire realistic sized DNN on a reticle-limited die.

One discovery in the field of energy-efficient computing is that it can often be sufficient to simply shift the activation; in other words, instead of multiplying X by 4, we simply left-shift it by 2. Our compute unit thus reduces from a multiplier-accumulator to a shifter-accumulator, where X_{kj} is shifted by $\log_2(W_{ik})$. This works best when our weight values are already **power-of-two (Po2)**, and some research has been conducted on making neural networks accurate when quantized to Po2 weights [8, 37], but the hardware works regardless of weight values. Because shifters are much smaller than multipliers, our computational density becomes much higher, allowing us to achieve a higher throughput for the same area cost. Most work done with shift accelerators is performed academically, and is run on FPGAs or CGRAs for cost-purposes; the general terminology for such an array is Neural Processing Unit or NPU, which refers to a systolic array of specific hardened computational units. We refer to these specific shifting array designs as "Shift NPUs," since they are systolic arrays with shift-and-accumulate compute units. For a detailed discussion of these approaches see Section 6.

Our key insight is that hardening design-time fixed Po2 weight values into rewiring-and-accumulate modules results in a significant jump in per-layer area efficiency. As we describe in detail in the next section, this both makes the spatial unrolling of a realistic size neural network feasible, and also allows enough area left over for supporting strategic flexibility.

3 Proposed Hardwired Accelerator Architecture

In our architecture, given in Figure 2, we load the initial inputs into on-chip memory using high-speed interconnect, run the image through our hardened convolution, and save the results in a small buffer that reflects the outputs of our feature extractor. By using power-of-two weight compression, which reduces the multiplication to rewiring when the convolution weights are fixed, we are able to fit our entire neural network architecture on-chip. The last layer – a fully connected layer for classification – is run in an on-chip NPU to maintain classification flexibility, and the outputs are

fed back to our main processor. Our chip's input is a series of HxW color images streamed onto the chip, and the output is a series of k-length vectors that correspond to softmax probabilities which are streamed back out from the chip. Our hardened convolution corresponds to the feature extractor of neural network architectures, while the on-chip NPU is the classifier layer that accesses the feature-extractor outputs (k_{fe}) stored in registers.

We begin by explaining the benefits of this hardcoded approach, explain each of the subcomponents' translation to hardware in greater detail, and conclude the section with a description of potential tradeoffs of the fixed-function design.

Using a hardened accelerator comes with several benefits, enumerated below.

3.0.1 No weight transfers from off-chip. Because our chip "stores" the convolution weights in the operation of its convolution engine, we do not need to continually fetch and store our filter weights from DRAM. Off-chip memory access comes with a hefty latency and energy pricetag; thus, by avoiding this bottleneck we can achieve both higher throughput and higher energy efficiency, a combination which is often a tradeoff in other settings.

3.0.2 No inter-layer I/O transfer. In addition to not needing to transfer weights from DRAM, we also do not suffer from the difficulty that other DNN accelerators face, namely, transferring the outputs of each layer either off-chip or into a streaming buffer. Because all layers of our design happen simultaneously on-chip, every layer's inputs and outputs do not need to be intermediately stored but propagate through the design. This only requires us to store the last outputs in an on-chip buffer, which is orders of magnitude smaller (k_{fe} instead of a double-buffer with both layer inputs and outputs i.e. $PQM + HWC$; for example, in MobileNetV2 this is 1280 elements compared to 802,816 elements to store Layer 2 Conv) and thus requires both less energy and area to store.

3.0.3 Fast cycle counts and throughput for entire model. The main benefit of our hardcoded design is its cycle count. Because we constant-propagate all of our convolution operations in parallel, and because each convolution operation involves spatial accumulation instead of accumulating over PEs, our entire feature extractor's latency reduces to several cycles. This allows extremely high throughput, on the order of millions of images per second (Section 5), making it a good design choice for users with high throughput demand.

3.0.4 Increased area efficiency. Because our convolutional layers are purely combinational, we greatly reduce the amount of area and energy required to calculate and store our elements (Section 5). In particular, our shifting-to-rewiring algorithm removes the need for multiplication or even shifters, but translates directly to rewiring, which uses negligible area and energy. This translates to immediate cost savings, allowing users to purchase a greater number of chips at a functionally equivalent running-price.

3.0.5 Sparsity has immediate benefits. There has been much research on increasing sparsity in convolutional neural networks without decreasing accuracy [34]. However, this has limited benefits in the actual hardware of most accelerators: sparse weight matrices can be compressed, but their reliance on compression schemes requires structured sparsity to run efficiently, which limits our search space for sparsity benefits. In addition, because traditional accelerators must still compute matrices of reduced size, the decrease in our compute cycles is actually sublinear (Section 5). In contrast, pruning an operation from a hardened, constant-propagated accelerator means removing its rewire-and-accumulate from the chip; this does not require compression schemes because the compression format is hardwired onto the chip, allowing for benefits at any pruning percentage, and the isomorphism between sparsity and hardware units dictates a linear reduction in area and energy.

3.1 Hardwired Convolution

Our neural network compression depends on quantizing our weights to power-of-two format. The traditional algorithm for a convolution uses floating-point 32-bit weights:

$$O_{pqm} = \sum_{r=0}^{R} \sum_{s=0}^{S} \sum_{c=0}^{C} W_{m,r,s,c} \cdot I_{p+r,q+s,c}$$

where a traditional accelerator involves a processor array of full 32-bit multipliers to compute the element-wise multiplication. How the accelerator divides up its multipliers for element computation depends on chosen stationarity, but involves time-multiplexing the same multipliers for multiple computations.

We rely on DeepShift's [8] quantized activations and weights, where our weights are power-of-two format. When we represent $W_{m,r,s,c}$ as 2^p for some integer p within bitshift range, the multiplication simplifies to a bit-wise shift:

$$W_{m,r,s,c}x = 2^{\mathrm{p}}x = \begin{cases} x << \mathrm{p} & \text{if p} > 0 \\ x >> \mathrm{p} & \text{if p} < 0 \\ x & \text{if p} = 0 \end{cases}$$

Each power-of-two quantization entails some individual rounding errors (Figure 3) but the network on the whole can achieve state-of-the-art precision for image classification [8].

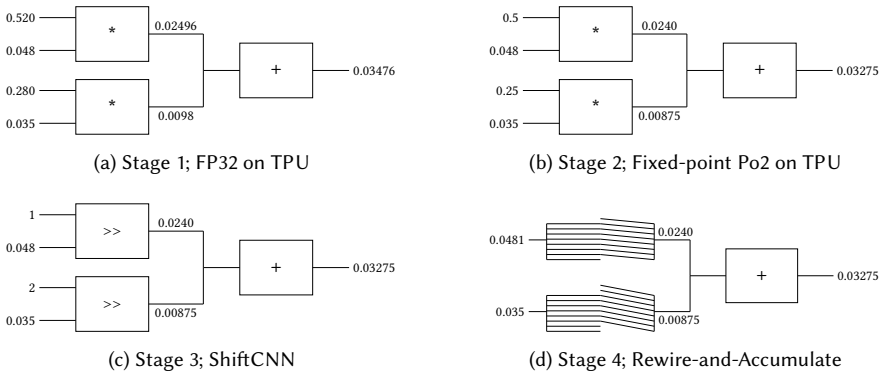
Previous works have utilized the hardware-friendly shifter operation to reduce area-size, where the chip setup is equivalent to a "Shift NPU" with processing arrays of variable shifters in place of multipliers [11, 30, 35]. Our crucial difference from this approach involves the realization that, at inference-time, the neural network's filter weights are fixed; therefore, if we can propagate all of our multipliers on-chip instead of time-multiplexing them, then each weight multiplication reduces to a fixed shifter, where the shifter is tied to the specific filter weight. However, this is equivalent to a simple rewiring, which takes no area at all. Moreover, there is no accuracy tradeoff to this calculation, because the result of rewiring is equivalent to the result of a fixed-function shifter, which is in turn equivalent to the result of a binary shifter set to a specific value at runtime.

Because we rewire instead of using multipliers, our area overhead for each element multiplication is nonexistent, so we can constant propagate all of our values to happen at the same time. The area cost for our convolution is solely due to the reduction of these elements into one output: this requires $RSC - 1$ adders, which may further simplify (i.e., addition of two right-shifted elements would require only a 7-bit adder instead of an 8-bit one).

To prevent overflow, we need an expanding adder tree such that our first layer is eight bits, our second layer is nine bits, and so on. Each layer halves the number of partial sums that need to be reduced, which halves the number of adders of that size required. We have RSC computations in each partial sum reduction to a single output O_{pqm}; our number of layers is the number of times we have to halve RSC before reducing to a single output, which is $log_2(RSC)$. We have $\frac{RSC}{2}$ 8-bit additions in the first layer to add all RSC elements together; in the successive layer we need $\frac{RSC}{4}$ 9-bit additions, and this number halves with each layer. Thus, our adder tree's area is proportional to the following formula:

$$\sum_{i=0}^{log_2(RSC)} (RSC \cdot 2^{-(1+i)}) \text{ (8+i)-bit adders.}$$

If we reduce our activation quantization to n-bits, this can have a significant saving in area and energy by reducing the size of our adders; we analyze this tradeoff in Section 5.2.



(a) Stage 1; FP32 on TPU

(b) Stage 2; Fixed-point Po2 on TPU

(c) Stage 3; ShiftCNN

(d) Stage 4; Rewire-and-Accumulate

Fig. 3. The four stages of multiplication compression. We begin with a full n-bit multiplier, which involves $O(n^2)$ full adders and $O(n)$ half-adders. A traditional accelerator time-multiplexes this multiplier, with the weights being refreshed for each compute. Stage two involves converting the weights to the nearest power-of-two, which incurs some accuracy reduction. Stage three replaces our multiplier with a bit shifter, which uses $O(n)$ multiplexers. Shift NPUs time-multiplex this shifter in the same way that traditional accelerators time-multiplex their multipliers. Finally, stage 4 is our novel approach: by constant-propagating all shifters on-chip and hardening their weight values, we convert each shifter into a simple rewiring, which uses no logic.

The large number of adders used in our constant-propagated design also influenced our choice of algorithms: another commonly used quantization scheme for neural networks is additive-power-of-two or APOT weights, where each weight is represented as an addition of several powers of two to smooth the values. Each power of two inside of the weight would be computed as a separate input rewiring in our design, which corresponds to an extra full-adder; thus, using APOT with two sub-elements for each weight would double the size of the design, while three sub-elements would triple it. Given the near-SOTA accuracy for Po2 quantization, we elected not to utilize APOT for this design.

In total we have PQM adder-trees for each layer, where the formula for the adder-trees' area is given above. The total number of elements is large, but as Table 4 shows, most adder-trees are fewer than a hundred microns and several hundred microns at maximum; thus, we will be able to fit our convolution on-chip for a carefully chosen model and pruning.

3.2 Batch Normalization

The second main computation in neural networks is batch normalization, which does a per-channel normalization of the inputs:

$$batchnorm(x_i) = \gamma\left(\frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}}\right) + \beta$$

where γ and β are learned scale and shift-parameters and μ_b, σ_b are the computed batch mean and variance, respectively. To compute these means and variances would require per-element subtraction, multiplication, and division, which would incur heavy area and power costs when performed in either a constant-propagated or time-multiplexed manner. However, because our chip is only used for inference, the mean and variance are not actually computed but are used from stored running-buffers, which stay constant; thus, we can fold the weights into the convolution:

$$batchnorm_{inf}(x_i) = \gamma\left(\frac{(Wx_i) - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}}\right) + \beta$$
$$= \left(\frac{\gamma W}{\sqrt{\sigma^2 + \epsilon}}\right) x_i + \left(\beta - \frac{\gamma\mu}{\sqrt{\sigma^2 + \epsilon}}\right)$$

which is the same as a biased convolution operation. PikeLPN [24] quantizes this batch normalization by quantizing $s = Q\left(\frac{\gamma}{\sqrt{\sigma^2+\epsilon}}\right)$ and $b = Q(\beta) - Q\left(\frac{\gamma\mu}{\sqrt{\sigma^2+\epsilon}}\right)$ and writes the batch normalization as $batchnorm(x) = x * s - b$, with the ability to fold the quantized batch norm into the convolution at runtime. Our quantization augments this approach with the additional constraint that our weights need to be quantized to Po2; therefore, we quantize all variables $\gamma, \beta, \sigma_b, \mu_b$ to fixed-point and additionally quantize γ and $\sqrt{\sigma^2 + \epsilon}$ to a Po2 representation. Since all weights in W are Po2, we can combine them together to get a single shifting convolution that corresponds to the combined variance-normalized convolution, and then add a fixed-point bias term to offset the mean. This requires one additional adder for each output term, or PQM additional adders per layer.

3.3 ReLU

Finally, we focus on our activation function. Several activation functions are commonly used in neural networks, including sigmoid, softmax, leaky ReLU, and Swish; we focus on ReLU, as it is the most common function and the cheapest to implement in hardware. ReLU keeps the value of its input if it is positive, otherwise it clamps it to zero:

$$ReLU(x) = \begin{cases} x & \text{if x} > 0 \\ 0 & \text{if x} \leq 0 \end{cases}$$

With a signed fixed-point input, we can perform this in hardware by inverting the most significant bit and anding it with the rest of the bits. Because this operation only requires two gates, it is very small in hardware, requiring only 2 cells and 0.1 μm^2 area in Asap 7nm; it must be performed once for each output, or PQM times per layer.

3.4 Flexible Classification from Final Layer

While our feature extractor is compressed and hardened on-chip, the last layer(s) of neural networks is a classifier, which is a fully-connected layer performing matrix-vector multiplication to compute the class logits for our input. If the feature extractor has k_{fe} outputs and can be written as a vector $X^{k_{fe}}$ and our network is a single matrix multiplication A that outputs k classes, this is an mc matrix multiplication

$$O^m = A^{m \times c} X^c.$$

To maintain flexibility and transfer learning capability, it is crucial that this layer is performed on an on-chip NPU, where the weights are kept in a local buffer and can be streamed from the processor if the user wants to perform a different task.

Leaving this last layer on-chip incurs negligible cost. While we explore the specific cycle count for MobileNetV2 in Section 5.2, the fact that this is a relatively small matrix-vector multiplication means we can perform this operation without incurring heavy area or latency costs. For instance, in MobileNetV2 the classification head is a 1000-by-1280 weight matrix multiplying a 1280-by-1 vector, which makes up only 0.4% of the MACs performed across all of the feature extractor layers. Our design involves our feature extractor activations being fed directly into the NPU, allowing

for an output-stationary dataflow with a cycle count proportional to k. In particular, we note in Section 5 that this cycle count is often comparable to the number of cycles required to load our input image data onto the chip, and thus by using pipelining we entirely amortize this cost. Both the weights and the inputs to the NPU are stored in local buffers, where the input buffer is the intermediate buffer used to store our feature extractor activations and the weights buffer is a $k \times k_{fe}$ element buffer that can also be accessed from the main memory bus. This allows the user to stream new transfer learning weights onto the chip, which are stored locally and allow the chip to perform the new image classification task. After we perform this final matrix multiplication, we gather the final output classification probabilities and stream them off-chip.

3.5 Tradeoffs of Hardwired Accelerators

Limited academic exploration of hardened CNN accelerators can largely be attributed to fears about the drawbacks of hardening a chip design, especially given the slow development time of ASICs and quick development time of new models. However, many of the initial drawbacks to the approach are not limiting:

3.5.1 The design is size-limited. Users who are interested in achieving accuracy at the cost of any other variable might be dissuaded by the limitation of fitting the network entirely on-chip. ResNet-50 achieves maximal classification accuracy among the CNN models, but even compressed using our Po2-rewiring scheme its sheer number of elements and layers greatly exceeds reticle limit. However, we argue that ResNet is not necessary to achieve good accuracy; Google's MobileNetV2 and V3 architectures both achieve within 5% accuracy with hundreds of times fewer operations, and remain the standard among lightweight models while also being competitive among full-scale models. Thus, the size-limitation to run lightweight models should not pose a downside for most use cases.

3.5.2 The design is architecture-specific. Another concern is that by creating hardware that only encodes a specific model, which is likely to become outdated by the time the chip is manufactured. However, we note that both MobileNetV2 and MobileNetV3 have been around since 2019 and remain along the size-accuracy pareto frontier. We believe that the long duration of these models eases these concerns and allows for the chip's production without obsolescence factors.

3.5.3 Quantization and sparsity required for sufficient compression. Finally, users might be concerned about the utilization of both Po2 quantization and sparsity in our model design. In particular, they might be concerned that Po2 quantized models cannot achieve SOTA for their use-cases, while other critics might argue that since the benefits of sparsity are so persuasive, there will be a race to continually sparsify the model, preventing it from ever being hardened for production. To the first case, we state that Po2 eight-bit quantized models have minimal accuracy loss when retrained on the relevant dataset [8], and we demonstrate in Section 5 that this also applies to transfer learning situations. To the second case, we acknowledge that our sparsity results may be surpassed; however, we claim that neural networks can only become so sparse before incurring an unacceptable accuracy loss, and that future attempts at pruning will not lead to instability in the specific die mask used long-term.

4 Training the Model

4.1 Neural Network Architecture Selection

To utilize our chip properly, we first need to carefully choose our model; using too large a neural network would prevent us from being able to harden the feature extractor on the chip, while using a small or obsolete model would limit our accuracy and use-cases. We performed an analysis of different popular CNN models, such as the MobileNet, VGGNet,

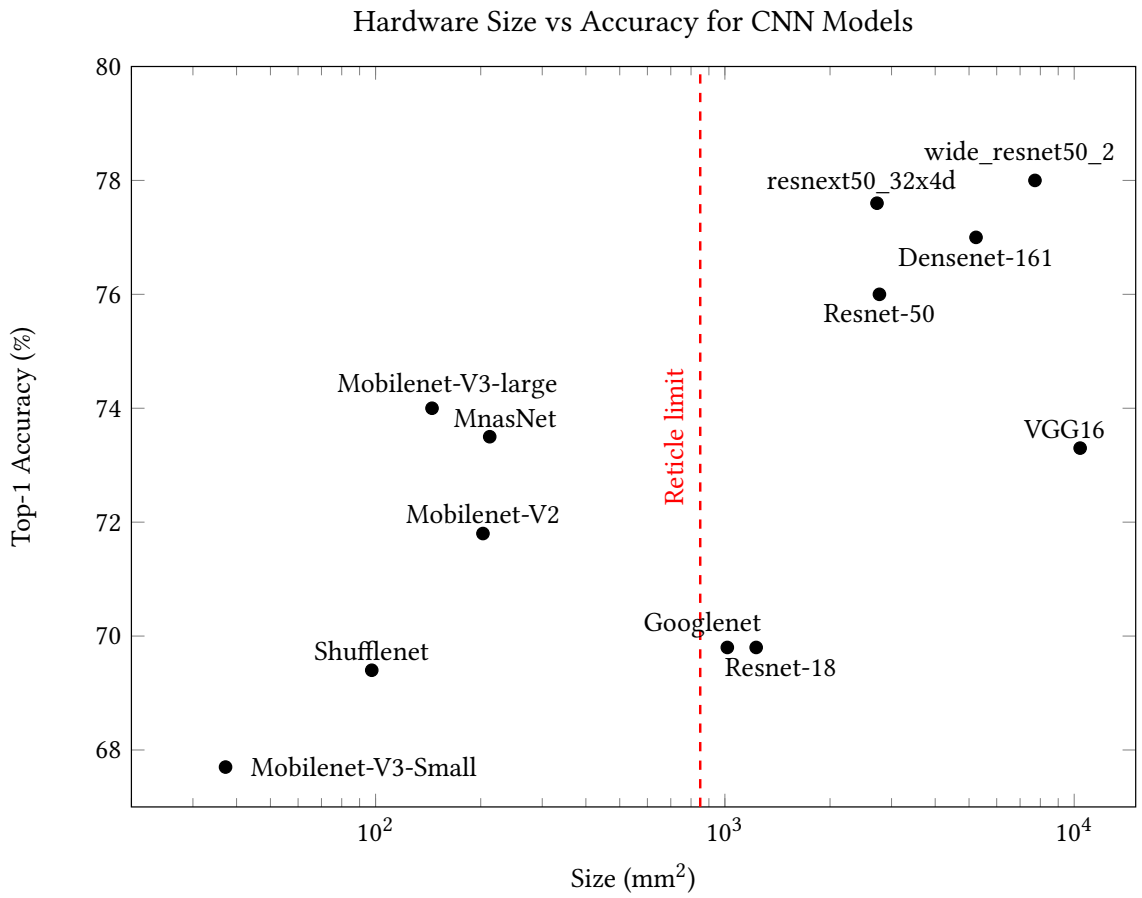


Fig. 4. Neural network architecture size when quantized to Q3.5 and compressed using our hardcoding procedure, compared to top-1 accuracy in pretrained model zoo. The MobileNet family gives a good tradeoff of chip size without sacrificing accuracy.

and ResNet families, and plotted their feature extractor size when hardened on ASIC against their top-1 accuracy from the pretrained model zoo (Figure 4).

As we can see, there is a tradeoff between size and accuracy that prevents us from hardening ResNet-50, a very deep and dense model, as its hardened size is far above the reticle limit. However, the MobileNet families achieve high accuracies while being orders of magnitude smaller than ResNet, making them a good choice for our analysis. In particular, MobileNetV2 [29] is conceptually easy to translate to hardware, as it consists only of convolutional layers, batch normalization, and ReLU activations while still achieving top-1 accuracy similar to VGG-19. While MobileNetV3-large achieves better accuracy than V2, its usage of Hardsigmoid and Hardswish activations requires lookup tables instead of the simple invert-and-and operation for ReLU, increasing both model complexity and area, such that MobileNetV2 remains the best candidate for this approach.

4.2 Quantization and Secondary Sparsification

For a given neural network, there are a variety of quantization approaches, including INQ [37], ABC-Net [20], and LogNN [21], which take different approaches to compression and training regimes. Our hardware accelerator requires a Po2-weights compression with minimal encoding of each multiplication to a single binary shift to minimize adders, while quantizing our activations allows our adders to have lower bitwidths; thus, we used DeepShift [8] as it provides all of these functionalities.

Existing research has shown that we can perform weight-Po2 quantization on Resnets and VGGNets with minimal Imagenet top-1 accuracy loss, and on MobileNetV2 with minimal CIFAR-10 top-1 accuracy loss [8]. We begin by quantizing pretrained MobileNetV2 to verify that the model's Imagenet top-1 accuracy loss is minimal. We also quantize our inputs to fixed-point implementation to fit on-chip; we assess accuracy for different input bitwidths. Because our weight shifting corresponds to input rewiring, there is no hardware savings from reducing our weight bitwidth to 5 bits as performed in DeepShift. Therefore, instead of reducing our weight bitwidth, we keep this value equal to the value of our input bitwidth; for example, an input quantization of 3 integer bits and 5 fraction bits, represented Q3.5, would have a weight bitwidth of 8. Our default quantization was Q3.5 as performed in DeepShift [8], and all learning parameters were kept the same between quantizations. Our fully classified layer was kept at FP32, as this will be run at full-precision on an on-chip NPU.

An additional, novel operation to DeepShift's experimental paradigm involved quantizing our batch normalization layers. As shown in Section 3.2, our accelerator involves folding the batch normalization into the convolution operation, which requires the batchnorm layer's running mean and shift variables to be fixed-point quantized and the running mean and scale variables to be Po2 fixed-point quantized. We run our analysis with these quantizations, where the Po2 bitwidths correspond to the weight bitwidths in the convolutional layers and the fixed-point quantizations correspond to the input bitwidths into the convolution.

After identifying the bitwidth-models within our accuracy threshold, we use sparsity as a secondary compression technique, and assess the amount of pruning that each model can achieve without severely reducing the accuracy. In particular, while we will be folding our batch normalization layers into the convolution when hardening our inference engine, it is sufficient to prune our convolutional layers to achieve this sparsity savings. Because our convolution weights become $\left(\frac{\gamma W}{\sqrt{\sigma^2+\epsilon}}\right)$ after folding, the batch-normalization factor $s = \frac{\gamma}{\sqrt{\sigma^2+\epsilon}}$ is a scalar multiplier onto our filter weights, and does not change which weights are smallest and should be pruned. Thus, we can prune just our convolutional layers and fold the batch normalization at runtime without consequences.

Because MobileNetV2's depthwise separable convolutions incur minimal area costs (Table 2), we do not sparsify these layers nor the first layer of our model. We begin pruning from the previous, quantized model version and train at an initial learning rate of 0.001 decreasing over time, with monotonically increasing sparsity.

5 Evaluation

We first analyze the statistics of our accelerator, breaking down the area, latency, and power of the chip's feature extractor, classifier layer, and on-chip buffers.

Methodology: We use Cadence Genus Synthesis Version 19.10-p002_1 with the Asap7nm PDK, mapping to gate-level logic and optimizing. Our last-layer NPU is based on SCALE-Sim's [28] verilog code. We use MobileNetV2 for our model, which takes in images of $224 \times 224 \times 3$ and has a feature extractor output size of 1280, which it classifies using a 1000×1280 matrix multiplication. We quantize MobileNetV2 to Q3.5 with Po2 weights, and train the model to 60%

sparsity with minimal accuracy loss (Section 5.2). We run SCALE-Sim in GEMM format for different NPU sizes and show the speedup factor for 2:4 weight encoding. Finally, we assess the flexibility of our model by employing transfer learning on other image datasets and comparing the results to transfer learning with the original MobileNetV2.

5.1 Main Result: Area, Throughput, and Latency

Table 2. Sub-component Area For Pruned MobileNetV2

Layer name	Size (mm^2)
Feature extractor	219
1024x1 on-chip NPU	0.24
On-chip buffers	0.42
Total (1 accelerator)	**220**
Total (4 accelerators)	880

Detailed area breakdown is shown in Table 2. Using 60% sparsity, our area for a Q3.5 feature extractor is 219 mm^2. On-chip buffers and our last layer NPU take up less than 0.5 mm^2. Running SCALE-Sim [28] with a general matrix multiplication of 1000x1x1280 MNK output-stationary format, our minimum achievable cycle count is 2278 cycles on a 1000x1 NPU array. Using SCALE-Sim's provided systolic array code, our area costs for this unit is 0.24 mm^2. Thus, each individual accelerator is **220 mm^2**. We can duplicate our accelerator four times before reaching reticle limit, for a total chip size of 880 mm^2.

In contrast, HaShiFix does not have this last layer NPU, and thus does not require either the multiplier cost or the 1280-element buffer used to separate the feature extractor from the classifier unit. However, as these are relatively small area costs compared to the size of our accelerator, our HaShiFix area is not significantly different from HaShiFlex.

Table 3. Comparison against SOTA Accelerators

Design	Throughput (im/s)	Latency (μs)	Area (mm^2)
HaShiFlex	$1.21 * 10^6$	3.3	880
HaShiFix	$4.0 * 10^6$	0.25	550
H100 GPU	$\approx 60,000$		814
Google TPU v4	100	$\approx 2,600$	600
GraphCore M2000	$\approx 10,000$	520	4 ×823

Table 3 shows an overall breakdown of throughput and latency for HaShiFlex and HashiFix derived from our methodology. Overall, we find our flexible HaShiFlex approach able to finish full forward passes through all layers on millions of images per second, an improvement of roughly 20.2× in throughput over fully programmable GPU solutions. In situations where no post-deployment adaptability is needed, this advantage increases to 67× for the fully-fixed weight HaShiFix solution.

Comparative Analysis: Numbers are hard to find for MobileNetV2 throughput and inference for datacenter TPUs and GPUs, but all metrics underperform by several orders of magnitude. Google's Edge TPU achieves 2.6 ms latency for unpruned eight-bit fixed-point MobileNetV2 [7], which is **800x** slower than our design. Running on a server TPU, we found inference throughput of 100 FPS for an INT8 MobileNetV2 with a batch-size of 32, which is **12000x** below our design. An NVIDIA H200 GPU can run ResNet-50v1.5 at INT8 at 65,045 images/sec [26]; while we cannot directly compare statistics for MobileNetV2, this indicates orders of magnitude less volume. Similarly, the GraphCore IPU-M2000

can achieve 0.52 ms latency and 9,404 images/sec throughput on Q16.16 ResNet-50v1.5 with configurations tuned specifically to optimize these parameters; even accounting for latency reduction for fixed-point 8-bit and MobileNet, both our throughput and latency perform 2 orders of magnitude better. Jiang et al. [14] created an FPGA accelerator specifically for Mobilenet-V2, which achieved 1910 FPS; our work achieves a **600x** better framerate. On MLPerf, a Qualcomm Hexagon (TM) NPU achieved 2,036.56 images/sec on MobileNetV4 [22], which is again **600x** less volume. The smallest reported image classification latency for a MobileNet on MLPerf [23] was Qualcomm's Snapdragon 8 Gen 3 Mobile HDK, which achieved 190 μs on MobileNetV1 0.25x, a much smaller model; even so, their latency is still **57x** larger than ours. Given the existing limitations of reticle area and interconnect speeds, our design is limited by memory bandwidth for a chip that only loads the image itself and outputs the final output; thus, it achieves the maximum achievable throughput and minimum latency for CNN inference.

5.2 Efficiency Improvement Details

Because these bottom-line numbers can be difficult to understand in isolation, we now engage in a thorough explanation of the source of the efficiency improvements.

- Our sparsity linearly impacts the size of our chip. For a chip of size 549 mm^2, a sparsity factor of s evaluates to a chip of size $a_{individ} = 549(1 - s)$.
- Our parallelization factor depends on chip size; given a reticle limit of 850 mm^2, the amount of possible parallelization is $k = \left\lfloor \frac{850}{a_{individ}} \right\rfloor$.
- NVIDIA's H100 GPU is 814mm^2 and has an interconnect speed of 900 GB/s. Interconnect scales linearly with area size, giving us interconnect speeds of $900 * \frac{\text{area}}{814} = 900 * \frac{k * a_{individ}}{814}$, but each accelerator can only access $\frac{1}{k}$ of the memory bandwidth, giving each accelerator a memory bandwidth of $\frac{900}{814} a_{individ}$ GB/s.
- Our chip operates at 1 GHz, limited by our NPU array; our feature extractor operates at a much slower speed, but does not need to be clocked at the same speed. Our per-chip interconnect corresponds to $\frac{900}{814} a_{individ}$ bytes of data bus.
- For each run, an accelerator needs to load one image and store one 1000-length vector of probabilities off-chip. Since our accelerator operates at Q3.5, we only need to load 8-bit image vectors, but our final classification probabilities are FP32, requiring 32 bits. The amount of data through memory bandwidth is $(224^2 * 3 + 1000 * 8)$ bytes per accelerator.
- Our NPU array requires 3300 cycles, regardless of sparsity or area sizes. Because our data input, feature extractor, and NPU are pipelined, our maximum latency is determined by the larger of 3300 cycles and the number of cycles to load our data. Below 65% sparsity, we are limited by NPU cycle count; above this sparsity, we are limited by data loading.
- Our latency directly corresponds to our cycle count; for 3300 cycles, we achieve **3.3 μ s latency**.
- Our throughput is equal to our parallelization factor multiplied by our individual accelerator throughput, which is $k \frac{10^9}{l}$ (Figure 5c). For 65% sparsity, we achieve **1.2 million images/sec**. Our maximum throughput is at 69% sparsity, where we achieve 1.24 million images/sec. Beyond this point, the effects of limited memory bandwidth outweigh the benefits of parallelization, and both throughput and latency degrade.

In contrast, HaShiFix is not limited by the 3300 cycle count, but only by the time to load the data. Thus, we calculate our cycles directly from parallelization and data load time, included in Figure 5c below. At 0% sparsity, we have the maximum chip size allowing for the most memory bandwidth; $a_{individ} = 549mm^2$, so our interconnect speed is 607

GB/s. With a 607-byte bus devoted to a single accelerator loading $(224^2 * 3 + 1000 * 8)$ bytes of data, this only takes 250 cycles, allowing for a latency of $0.25\mu s$, 13 times lower than our HaShiFlex latency. Because we can't parallelize, our throughput becomes $\frac{10^9}{250}$ = **4 million** images/sec, almost four times higher than HaShiFlex's design.

5.3 Impact of Model Compression

Given that our model is originally unpruned and only quantized to eight-bit, we assess the extent to which we can compress our model along both quantization and sparsity axes, as well as the size of the model in each case.

Table 4. Hardened Convolution sizes (μm^2) for different input bitwidths in Asap 7nm.

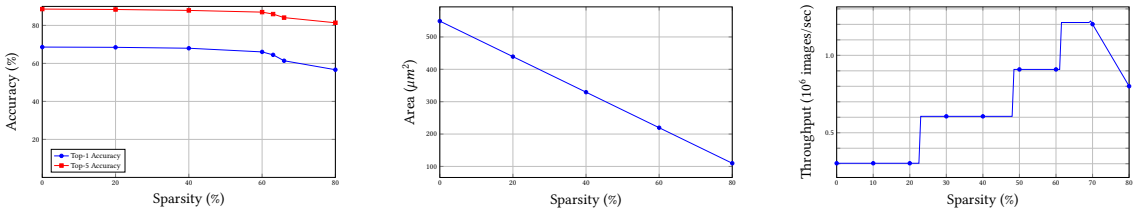
Convolution	5-bit	6-bit	7-bit	8-bit
3x3x3	27.3 (54.6%)	35.9 (71.8%)	43.5 (87%)	50.0
3x3 (pw)	1.0 (100.0%)	1.0 (100.0%)	1.0 (100.0%)	1.0
1x1x16	16.4 (55.8%)	21.0 (71.4%)	25.0 (85%)	29.4
1x1x32	33.3 (54.6%)	43.7 (71.6%)	50.0 (82%)	61.0
1x1x64	72.6 (57.6%)	88.2 (70.0%)	106.4 (84.4%)	126.0
1x1x320	362.9 (57.4%)	450.7 (71.2%)	543.2 (85.9%)	632.6
MAC Unit	16.8 (53.9%)	21.3 (68.3%)	25.9 (83.0%)	31.2

Table 5. MobileNetV2 Weight bit (WB) Po2 Quantization and Input bit (IB) Quantization Versus Accuracy

Input Bitwidth	Top-1 Accuracy (%)	Top-5 Accuracy (%)
Original	72	90
WB 8, IB Q3.5	68	88
WB 7, IB Q3.4	66	87
WB 7, IB Q2.5	55	80
WB 6, IB Q3.3	56	80
WB 6, IB Q2.4	22	45
WB 5, IB Q3.2	0.23	1.1

Running our convolution modules in Genus with the Asap7 PDK, we get the area costs for an entire hardened convolution unit (Table 4). The majority of our area costs are in pointwise convolutions, which are also our best candidates for compression. Reducing our activations from 8-bit to 7-bit can provide area savings of 15%, while compressing to five bits is roughly half the area cost. However, when running MobileNetV2 with different input bitwidths, our accuracy degradation quickly becomes infeasible. Notably, our results degrade much more quickly than in DeepShift [8] because we quantize our activations as well as our weights, as activation quantization is crucial for reducing our adder bitwidths, while weight quantization (due to being simple rewiring) does not affect our area metrics. We find a sharp accuracy decline between Q3.5 and Q3.4 quantizations and all other quantizations; in addition, only Q3.5 is within a 5% accuracy threshold. Thus we use Q3.5 as our baseline quantization scheme.

Pruning is particularly beneficial for our chip, since sparsity level linearly decreases our area (Figure 5b). It is therefore critical to identify the maximal amount of pruning possible before our accuracy decreases. Running incremental pruning of 20%, with 30 layers of post-pruning retraining and with each pruning occurring on top of the previous fully-retrained model, we find that we can prune to 60% before significantly degrading our accuracy. Running a secondary, more fine-grained pruning of 3% incremental pruning with 10 epochs retraining from 60% to 69% sparsity threshold, we find that our accuracy degrades quickly after reaching 60% sparsity, so we utilize this value for our hardwired design. This



(a) Accuracy vs Sparsity (b) Area vs Sparsity (c) Throughput vs Sparsity

Fig. 5. Analysis of the effects of pruning on accuracy, area, and throughput. Accuracy analysis is taken from pruning our Q3.5 MobileNetV2 model with incremental sparsity of 20% and 30 epochs of retraining. Area analysis is taken from translating the resulting sparse convolution modules into Genus with Asap7nm. Throughput analysis comes from Section 5.1 and depends on achievable interconnect speed and parallelization capacity.

level of sparsity places us at 219 mm^2, which allows us a four-times parallelization factor and corresponding throughput increase.

In contrast, GPUs can only compress in a 2:4 weight compression strategy, which preserves two weights out of every four elements, or a 50% compression. Systolic arrays in general are often not built to take advantage of the small convolution sizes in MobileNetV2; TPU v4, for instance, is built around 128x128 systolic arrays and does not dynamically resize them, meaning that the arrays are already underutilized for smaller pointwise convolutions such as $1 \times 1 \times 24$. However, when we run the layers in SCALE-Sim [28] on a 128-by-128 weight-stationary config file, mirroring the TPU systolic arrays, we do find that the array can take advantage of 2:4 weight encoding. When we compare the cycle counts for each layer of MobileNetV2 against the same layer when the inner dimension is halfsize, $W^{M \times \frac{RSC}{2}} X^{\frac{RSC}{2} \times PQ}$ compared to $W^{M \times RSC} X^{RSC \times PQ}$, we find an average cycle count reduction to 83% of the original cycle count across the different layers, resulting in 60% of the total cycles. Depending on the specific layer choice, therefore, we can achieve linear savings on layers like 1x1x160 and 1x1x24, but fail to achieve cycle savings on badly tiled layers such as 1x1x144 or 1x1x192. In general, we note that while neural networks can be carefully constructed to be responsive to 2:4 weight encoding, the overall savings are often sublinear to the amount pruned.

In general, our hardware design is far more friendly to pruning than other accelerators: because pruning a weight from a layer directly corresponds to removing its adder from the reduction tree, our hardware demonstrates benefits when we prune to any amount. As we linearly reduce our area with sparsity, we can increasingly parallelize our accelerator, with no drawbacks to latency since we continue to be compute-bound by our individual last-layer NPUs.

5.4 Post-Deployment Flexibility and Accuracy

In order to make our model competitive against other inference accelerators, it is crucial that we remain able to adapt our feature extractor to different related image classification tasks. Without this capability, our accelerator's use-cases would be limited to baseline ImageNet classification; however, by retraining our last layer we can demonstrate successful transfer learning onto new datasets.

In particular, we compare our quantized, sparsified version of MobileNetV2 to the original MobileNetV2 for accuracy in transfer learning the CIFAR-10 and CIFAR-100 datasets. We note, crucially, that HaShiFix has a hardened classifier layer, making it impossible to perform transfer learning when the task is different from ImageNet; HaShiFlex's last layer flexibility allows it to be useful across a much broader range of tasks.

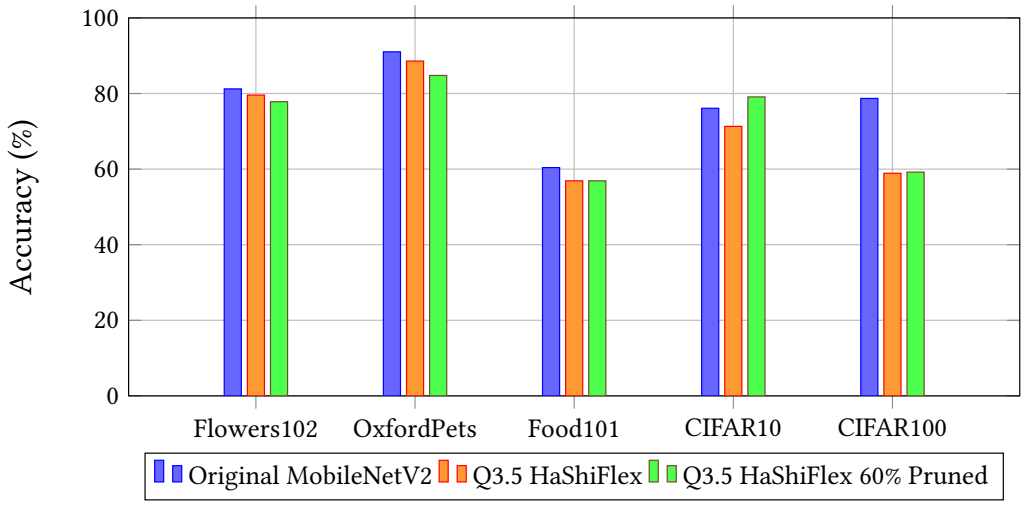


Fig. 6. Transfer Learning Accuracy Across Image Classification Datasets. All models are MobileNetV2 with fixed input dimension and frozen except for the last layer. Original is pretrained from model zoo, Quantized is our 8-bit Po2 MobileNetV2 without sparsity, and Sparse is pruned to 60% sparsity.

Our analysis is performed on the original FP32 MobileNetV2, the Q3.5 HaShiFlex model without pruning, and the Q3.5 HaShiFlex model pruned to 60% sparsity. The last layer is kept at FP32 precision and fully programmable, while all other layers are frozen. We chose Flowers102 [25], OxfordPets [27], and Food101 [1] as three representative datasets of transfer learning regimes, with high input resolution that benefit strongly from CNN's feature extraction capabilities. In addition, we included CIFAR10 and CIFAR100 [16] as Transfer Learning regimes, since their low resolution can demonstrate the limits of nonresizable neural networks. We retrained our models for 20 epochs with an initial learning rate of 0.01 and a step size of 5 epochs.

We see that our hardened model maintains sufficient flexibility in its last layer to learn new image classification tasks. All three variants suffer significantly for CIFAR10 and CIFAR100, since these are only 32-by-32 pixel images and our accelerator can only work with 224-by-224 inputs; thus, rescaling the image to this size causes most learned spatial features to be rendered useless. However, we believe that there are few cases when images will need to be processed at such low resolutions, and our datasets Oxford Pets and Stanford Dogs occur in variable image sizes, demonstrating that our network is still effective at learning from downsampled images.

Thus, we can see that our accelerator achieves throughput speedups of 400-600 relative to other hardware, with latency reduction on the order of 1000x. This makes it an interesting alternative for high-throughput and low-latency inference environments.

6 Related Work

6.1 Shifter-Based Hardware Accelerators

In 2017 Godovskiy and Rigazio introduced ShiftCNN [9], which proposed increasing hardware computational efficiency by using power-of-two weights in order to remove the multipliers and replace them with shifters. Furthermore, ShiftCNN employed aggressive post-training quantization that enabled pre-computation of all possible terms in the output tensor

based on the weights. Although this work was never formally peer reviewed, we include it because it had a large influence on the remaining works in this section.

Hsieh et al. propose a multiplier-less CNN accelerator [11] based around programmable RAM-based lookup tables whose sizes are limited by using a custom 4-bit quantization format called Signed Digit 4 (SD4). Beyond post-training quantization, the authors developed several techniques to aid in the challenges of in-training quantization, including compensating for gradient deltas that were notably smaller than the SD4 format's range would encode.

Song et al. propose a "universal shift" CNN accelerator [30] based on 8-bit activations and 4-bit power-of-two weights using lookup tables. Training is accomplished using shifted representation in-training, with post-training quantization.

Xu et al. propose a non-uniform codebook quantization using Huffman encoding [35] that can be implemented via lookup tables. They accelerate this pattern using a hierarchical systolic array of 14x14 PEs, where each PE consists of a 5-element 1D systolic shifter.

6.2 Multiplication-Free Neural Networks

Inspired by hardware accelerator efficiency gains, a category of work looks to leverage those gains by exposing the lack of multiplication operation directly to the neural network.

ShiftNet [32] described CNN convolutions by enumerating all the possible "shift matrices" that describe the shift directions of a layer. This allowed all multiplications to be replaced with pointwise memory transfers. Unfortunately, this made memory bandwidth a bottleneck. The Sparse Shift Layer [5] attempts to reduce this by eliminating ineffectual shifts, particularly when paired with quantization.

DeepShift [8] is a software-only neural network that uses shift-based techniques directly in the architecture of the layers. Additional contributions included reducing the number of shifts in a convolutional layer from ShiftCNN's 2-3 to 1, and extending the technique to linear layers. Additionally, DeepShift demonstrated that training power-of-two weights from scratch was possible, in addition to post-training quantization.

ShiftAddNet [36] uses separate shift and add layers, and demonstrated that the anchor-weight phenomenon can exist in shifting layers as well as traditional multiplied weights. AddressNet [10] specializes these into three primitives called "channel shift," "address shift," and "shortcut shift" that have efficient GPU implementations.

In contrast, AdderNet [2] demonstrated that shifts could be removed altogether by maximizing emphasis on addition operations, replacing the convolutional dot-product with calculation of the ℓ_1 distance between filter and input.

PikeLPN [24] is a network based around the insight that non-quantized elementwise operations such as activation functions become the bottleneck. It proposes a technique called QuantNorm to quantize batch norm layers that we leverage and expand upon.

6.3 Shifter-Based Quantization

Finally, we discuss a survey of methods aimed at improving accuracy for power-of-two-based quantization accuracy. We include this work as an overview of techniques that demonstrate the feasibility of our approach, and could be combined with our insights as future work.

DenseShift [18] combines power-of-two quantization with sparse pruning by proposing zero-free shifting. The resulting weights are densely encoded without zeros (which is technically not a power-of-two). Accuracy is recovered using a sign-scale decomposition technique during training.

JumpingShift [12] proposes an alternative scheme called Jumping Logic Quantization wherein a new coefficient is added to power-of-2 exponents, effectively "jumping" over regions in the quantization range, and also demonstrate removing zeros to increase shifter utilization without loss of accuracy.

Global sign-based network quantization (GSNQ) [13] assigns different ranges based on the sign bit of the weights during training. The authors demonstrate competitively high accuracy by retraining CNNs both inter-layer and intra-layer levels, effectively allowing later layers to compensate for pruned damage to earlier layers.

S^3 [19] observes that shift-based networks are highly sensitive to weight initialization during training, and can further suffer from vanishing gradients or sign-bit freezing due to short bit widths. They propose a sign-sparse-shift decomposition representation that improves learning rate and decreases initialization sensitivity.

7 Discussion

Our novel approach to full constant propagation of large neural networks allows us to reduce our area costs in significant ways. Crucially, because all weights exist on-chip, we can utilize power-of-two representation to convert each necessary multiplication into a fixed bit shifter and thus ultimately to rewiring. Thus, our area costs only come from the adder trees necessary for partial sum accumulation of outputs, and can fit in 220 mm^2 for a lightweight, sparse network such as sparsified MobileNetV2. Our approach allows for complete input multicasting and reuse, and eliminates the need to transfer any data on- or off-chip besides the input data, thus achieving the theoretical maximum inference speed on hardware.

Because of the state-of-the-art accuracy results of MobileNets even when these models are sparse, we are able to reduce our inference engine to fit comfortably on-chip and to allow for easy parallelization for increased throughput. Unlike GPUs and TPUs, which approach the reticle limit and still don't achieve relatively high model throughput, our design opens the possibility for chiplets that compute different tasks with the same input image. In particular, future work encourages us to focus on multi-task learning encoders, as a hardened encoder can run a variety of different tasks without much area increase from different decoder heads, greatly increasing flexibility.

Other possible avenues for future exploration include hardening different model architectures, such as transformers, as well as increasing task flexibility by embedding Low-Rank Adaptation matrices into the hardened convolutional layers. For actual chip production, there are still open questions to resolve, including making the chip robust to glitches. By providing our results as well as our torch-to-HDL libraries, we hope to foster community discussion and adoption.

References

[1] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. 2014. Food-101 – Mining Discriminative Components with Random Forests. In *European Conference on Computer Vision*.

[2] Hanting Chen, Yunhe Wang, Chunjing Xu, Boxin Shi, Chao Xu, Qi Tian, and Chang Xu. 2019. AdderNet: Do We Really Need Multiplications in Deep Learning? *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019), 1465–1474.

[3] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. *SIGARCH Computer Architecture News* 42, 1 (Feb. 2014), 269–284.

[4] Weihan Chen, Peisong Wang, and Jian Cheng. 2021. Towards Mixed-Precision Quantization of Neural Networks via Constrained Optimization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 5350–5359.

[5] Weijie Chen, Di Xie, Yuan Zhang, and Shiliang Pu. 2019. All You Need Is a Few Shifts: Designing Efficient Convolutional Neural Networks for Image Classification. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 7234–7243.

[6] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 367–379.

[7] Coral by Google. 2025. Edge TPU Benchmarks. https://coral.ai/docs/edgetpu/benchmarks/. Accessed: 2025-04-11.

[8] Mostafa Elhoushi, Zihao Chen, Farhan Shafiq, Ye Henry Tian, and Joey Yiwei Li. 2021. DeepShift: Towards Multiplication-Less Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2021, virtual, June 19-25, 2021*. Computer Vision Foundation / IEEE, 2359–2368.

[9] Denis A. Gudovskiy and Luca Rigazio. 2017. ShiftCNN: Generalized Low-Precision Architecture for Inference of Convolutional Neural Networks. *ArXiv* abs/1706.02393 (2017).

[10] Yihui He, Xianggen Liu, Huasong Zhong, and Yuchun Ma. 2019. AddressNet: Shift-Based Primitives for Efficient Convolutional Neural Networks. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*. 1213–1222.

[11] Ming-Hang Hsieh, Yu-Tung Liu, and Tzi-Dar Chiueh. 2021. A Multiplier-Less Convolutional Neural Network Inference Accelerator for Intelligent Edge Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 11, 4 (2021), 739–750.

[12] Longxing Jiang, David Aledo, and Rene van Leuken. 2023. Jumping Shift: A Logarithmic Quantization Method for Low-Power CNN Acceleration. In *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–6.

[13] Tao Jiang, Ligang Xing, Jinming Yu, and Junchao Qian. 2024. A hardware-friendly logarithmic quantization method for CNNs and FPGA implementation. *Journal of Real-Time Image Processing* 21, 4 (June 2024), 11 pages.

[14] Weixiong Jiang, Heng Yu, and Yajun Ha. 2023. A High-Throughput Full-Dataflow MobileNetv2 Accelerator on Edge FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 5 (2023), 1532–1545. https://doi.org/10.1109/TCAD.2022.3198246

[15] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. Association for Computing Machinery, New York, NY, USA, 1–12.

[16] Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning multiple layers of features from tiny images*. Technical Report 0. University of Toronto, Toronto, Ontario. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[17] Yann LeCun and Yoshua Bengio. 1998. *Convolutional networks for images, speech, and time series*. MIT Press, Cambridge, MA, USA, 255–258.

[18] Xinlin Li, Bang Liu, Rui Heng Yang, Vanessa Courville, Chao Xing, and Vahid Partovi Nia. 2023. DenseShift : Towards Accurate and Efficient Low-Bit Power-of-Two Quantization . In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE Computer Society, Los Alamitos, CA, USA, 16964–16974.

[19] Xinlin Li, Bang Liu, Yaoliang Yu, Wulong Liu, Chunjing Xu, and Vahid Partovi Nia. 2021. S3: sign-sparse-shift reparametrization for effective training of low-bit shift networks. In *Proceedings of the 35th International Conference on Neural Information Processing Systems (NeurIPS '21)*. Article 1115, 12 pages.

[20] Xiaofan Lin, Cong Zhao, and Wei Pan. 2017. Towards Accurate Binary Convolutional Neural Network. arXiv:1711.11294 [cs.LG] https://arxiv.org/abs/1711.11294

[21] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. 2016. Convolutional Neural Networks using Logarithmic Data Representation. arXiv:1603.01025 [cs.NE] https://arxiv.org/abs/1603.01025

[22] MLCommons. 2025. MLPerf Inference: Mobile Benchmark Results. https://mlcommons.org/benchmarks/inference-mobile/. Accessed: 2025-04-11.

[23] MLCommons. 2025. MLPerf Inference: Tiny Benchmark Results. https://mlcommons.org/benchmarks/inference-tiny/. Accessed: 2025-04-11.

[24] Marina Neseem, Conor McCullough, Randy Hsin, Chas Leichner, Shan Li, In Suk Chong, Andrew Howard, Lukasz Lew, Sherief Reda, Ville Rautio, and Daniele Moro. 2024. PikeLPN: Mitigating Overlooked Inefficiencies of Low-Precision Neural Networks. *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2024), 15996–16005. https://api.semanticscholar.org/CorpusID:268819385

[25] Maria-Elena Nilsback and Andrew Zisserman. 2008. Automated Flower Classification over a Large Number of Classes. In *Indian Conference on Computer Vision, Graphics and Image Processing*.

[26] NVIDIA Corporation. 2025. AI Inference Performance Benchmarks. https://developer.nvidia.com/deep-learning-performance-training-inference/ai-inference. Accessed: 2025-04-11.

[27] Omkar M. Parkhi, Andrea Vedaldi, Andrew Zisserman, and C. V. Jawahar. 2012. Cats and Dogs. In *IEEE Conference on Computer Vision and Pattern Recognition*.

[28] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2020. A systematic methodology for characterizing scalability of DNN accelerators using SCALE-sim. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 58–68.

[29] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation. *CoRR* abs/1801.04381 (2018). arXiv:1801.04381 http://arxiv.org/abs/1801.04381

[30] Qingzeng Song, Weizhi Cui, Liankun Sun, and Guanghao Jin. 2024. Design and Implementation of a Universal Shift Convolutional Neural Network Accelerator. *IEEE Embedded Systems Letters* 16, 1 (2024), 17–20.

[31] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 8612–8620.

[32] Bichen Wu, Alvin Wan, Xiangyu Yue, Peter Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. 2018. Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

[33] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. 2018. Mixed precision quantization of convnets via differentiable neural architecture search. *arXiv preprint arXiv:1812.00090* (2018).

[34] Sheng Xu, Anran Huang, Lei Chen, and Baochang Zhang. 2020. Convolutional neural network pruning: A survey. In *2020 39th Chinese control conference (CCC)*. IEEE, 7458–7463.

[35] Weihong Xu, Zaichen Zhang, Xiaohu You, and Chuan Zhang. 2018. Efficient Deep Convolutional Neural Networks Accelerator without Multiplication and Retraining. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 1100–1104.

[36] Haoran You, Xiaohan Chen, Yongan Zhang, Chaojian Li, Sicheng Li, Zihao Liu, Zhangyang Wang, and Yingyan Lin. 2020. ShiftAddNet: a hardware-inspired deep network. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NeurIPS '20)*.

[37] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights. In *International Conference on Learning Representations,ICLR2017*.