*Appears in the Proceedings of the 35th ACM International Conference on Supercomputing (ICS 2021)*

# Accelerating DNNs Inference with Predictive Layer Fusion

MohammadHossein Olyaiy
mohamadol@ece.ubc.ca
University of British Columbia

Christopher Ng
chris.ng@ece.ubc.ca
University of British Columbia

Mieszko Lis
mieszko@ece.ubc.ca
University of British Columbia

## ABSTRACT

Many modern convolutional neural neworks (CNNs) rely on bottleneck block structures where the activation tensor is mapped between higher dimensions using an intermediate low dimension, and convolved with depthwise feature filters rather than multi-channel filters. Because most of the computation lies in computing the large dimensional tensors, however, such networks cannot be scaled without significant computation costs.

In this paper, we show how *fusing* the layers inside these blocks can dramatically reduce the multiplication count (by 6–20×) at the cost of extra additions. ReLU nonlinearities are predicted dynamically, and only the activations that survive ReLU contribute to directly compute the output of the block. We also propose FusioNet, a CNN architecture optimized for fusion, as well as Archon, a novel accelerator design with a dataflow optimized for fused networks.

When FusioNet is executed on the proposed accelerator, it yields up to 5.8× faster inference compared to compact networks executed on a dense DNN accelerator, and 2.1 × faster inference compared to the same networks when pruned and executed on a sparse DNN accelerator.

## CCS CONCEPTS

• **Computer systems organization** → **Neural networks**.

## KEYWORDS

Neural networks, hardware accelerators, hardware-software codesign, ReLU prediction, bottleneck blocks, edge computing

**ACM Reference Format:**
MohammadHossein Olyaiy, Christopher Ng, and Mieszko Lis. 2021. Accelerating DNNs Inference with Predictive Layer Fusion. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3447818.3460378

## 1 INTRODUCTION

Modern mobile convolutional neural networks (CNNs) [16, 17, 32, 37, 42] rely on variants of "bottleneck" blocks to decrease the computational cost. Instead of conventional convolutions, these blocks employ depthwise convolutions to extract features and pointwise convolutions to distribute information across multiple channels. Typically, each bottleneck block consists of a pointwise layer that maps from few to many channels, a depthwise layer that detects features in each channel, and another pointwise layer that again significantly reduces the channel count. For example, in Figure 1, layer *(a)* expands the feature map from 64 to 384 channels, layer *(b)* detects features in each of the 384 channels separately, and layer *(c)* reduces the tensor back to 64 channels.

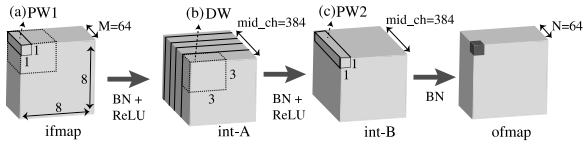



**Figure 1: MobileNet v2 block structure. The residual connection between the input and output of the block has been omitted for clarity.**

Together, the three stages of the bottleneck take 3.3 million multiply-accumulate (MAC) operations. Observe, however, that the overall operation maps a 64-channel tensor to another 64-channel tensor, despite the expansion to 384 channels as an intermediate step. Were it somehow possible to perform this operation directly as a linear composition of the three layers in the bottleneck, the number of MACs could be decreased, in this case to 30% less than the original value in the running example.

However, this straightforward linear composition is of course not possible because of the rectified linear unit (ReLU) nonlinearities [29] between the layers.

In this paper, we propose a layer composition technique that overcomes this challenge. Our approach relies on the observation that each activation after the final bottleneck layer is a linear combination of its components from all layers, with some components ignored (i.e., equal to zero) because of the ReLU filter. We therefore dynamically *predict* the ReLU outcome (zero or pass) during inference, and reconstruct the effective activation by adding together the components that would have survived the ReLU mask.

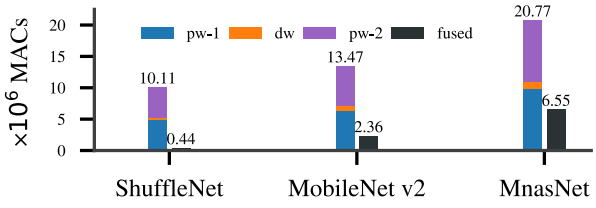



**Figure 2: Number of MAC operations in bottleneck layers from three CNNs, when executed normally as three layers (left) vs. as one fused layer (right); pw=pointwise layers; dw=depthwise layers.**


*ICS '21, June 14–17, 2021, Virtual Event, USA*

ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460378

Figure 2 quantifies the potential of this technique in terms of the number of million MACs needed to process a bottleneck block from three representative CNNs, both when all three layers are executed separately, and when the bottleneck is fused into a single layer with ideal ReLU predictions. The potential is significant — an order of magnitude — and promising even if savings will decrease with realistic ReLU prediction and effective activation reconstruction.

Recomputing the final activations this way saves many multiplication operations. However, it presents three new challenges: (a) the ReLU activation filters must be predicted dynamically; (b) the final weight for each feature map must be dynamically reconstituted by adding many sub-components, but *only* those that survived their ReLU activations; and (c) layer fusion significantly increases the model size as combinations of weights are multiplied out.

To overcome these challenges, we propose an end-to-end co-design approach that combines changes in the deep neural network (DNN) architecture as well as a novel hardware architecture suitable for executing fused DNNs. Specifically, we make the following contributions:

First, we observe that ReLU acts as a mask filter applied to an otherwise linear composition. Thus, if the ReLU pass/zero mask were known statically, the output feature map (ofmap) that results from multiple layer applications — such as that in a bottleneck block — could be treated as one combined layer, where the effective weights are different combinations of the weights from each layer combined depending on which activations survive the ReLU mask. Critically, because weights are known, the multiplication can be done *offline*, leaving only addition during actual inference (cf. Section 3). This saves a significant number of multiply operations (cf. Figure 2), at the cost of increasing the number of additions required.

Second, we develop a DNN hardware accelerator architecture that (a) predicts the ReLU mask accurately but at a fraction of the computation cost, and (b) uses the predicted mask to filter and dynamically accumulate the pre-computed fused weights.

Third, to address the intractable growth of fused weights, we observe that fusing previously pruned layers significantly increases the sparsity in fused weights. We use this observation to dramatically reduce the number of fused weights.

Fourth, we analyze different flavours of bottleneck blocks in state-of-the-art CNNs and create a new flavour that better takes advantage of layer fusion.

To the best of our knowledge, this is the first work to analyze true layer fusion (as opposed to simply using on-chip storage to pass activations through a pipeline of layers [4, 11], or merging batch normalization or ReLU operations with preceding convolutions). Our experiments show that, when maintaining similar accuracy, our techniques yield up to 5.8 × inference speedup on compact networks executed on a dense DNN accelerator, and 2.1 × speedup when these networks are pruned and executed on a sparse DNN accelerator.

## 2 BACKGROUND

In this section, we review how bottlenecks are used in modern networks, and then sketch the architecture of existing accelerators.

### 2.1 Bottleneck Blocks

Many recent works [e.g., 16, 17, 32, 37, 42] employ variants of *bottleneck blocks* to design more computationally-efficient CNNs. These blocks combine two ideas: (a) factoring a convolution layer into pointwise and depthwise layers, and (b) reducing the computation by turning the pointwise layer into a bottleneck structure.

**Depthwise (DW) convolutions.** Traditional CNNs rely on what we refer to as *full* convolutions: the kernel that produces each output activation has a dimension that corresponds to all input channels, and is convolved with all input channels at once. In contrast, the kernel in a *depthwise* convolution (illustrated as (b) in Figure 1) lacks this input channel dimension, and is applied separately to each input channel. This takes many fewer MACs: for example, a $3 \times 3$ regular convolution on an $8^2 \times 384$ input feature map (384 being the number of input channels) needs $3^2 \times 8^2 \times 384^2$ MACs to produce 384 output channels, while a depthwise convolution only needs $3^2 \times 8^2 \times 384$ MACs to produce an ofmap with the same dimensions, 384× fewer MACs than the regular convolution.

**Pointwise (PW) convolutions.** Performing depthwise convolutions, however, does not allow features to consider information from different input channels. To address this, pointwise $1 \times 1$ convolution layers ((a) and (c) in Figure 1) are used together with the depthwise layers to allow cross-layer information exchange. In effect, depthwise layers learn to extract channel-wise features, while pointwise layers learn to gather information from features in different channels and combine them.

**Bottleneck block structure.** Pointwise convolutions are still costly, as they gather information from a large number of channels and produce high-dimensional tensors. To optimize this computation, two separate pointwise layers can work in a bottleneck fashion; one of the pointwise layers (PW2 in Figure 1) maps the high-dimensional input to a low-dimensional tensor and the other PW layer (PW1 in Figure 1) maps this back to a high-dimensional tensor. We refer to the number of the channels in expanded block as *mid_ch*, and the ratio of the high dimension to the low dimension used in a bottleneck structure as $T$. For example, applying a pointwise convolution on an ifmap of size $8 \times 8$ with 384 channels to produce an ofmap of the same dimensions needs $8^2 \times 384^2 = 9,437,184$ operations. However, if we use a bottleneck structure with $T = 6$, we only need $8^2 \times 384 \times 64 \times 2 = 3,145,728$ operations, 3× fewer than in the original case.

**Example.** Figure 1 shows how DW and PW convolutions are employed in the MobileNet v2 [32] building blocks. The bottleneck block transforms an $8^2 \times 64$ input activation tensor into another $8^2 \times 64$ activation tensor as the output. The processing occurs in three distinct steps:

- First, the input activations tensor is projected into an $8 \times 8 \times 384$ intermediate tensor (*int-A*) by applying 384 pointwise convolutions, corresponding to an expansion factor of $T = 6$; this is followed by batch normalization [18] and ReLU as usual. This is shown in Figure 1a.
- Next, a depthwise $3 \times 3$ convolution is applied to each of the 384 channels separately, again followed by batch normalization and ReLU layers. this results in another $8 \times 8 \times 384$ intermediate tensor (*int-B*). This is shown in Figure 1b.

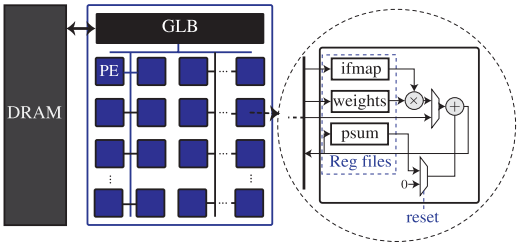



**Figure 3: A typical 2D-PE-array DNN accelerator: each PE has local buffers for various data types, as well as a MAC unit. A global buffer (GLB) acts as an intermediate level of storage between the PEs and DRAM to exploit data reuse.**

- Finally, the number of channels is reduced from 384 back to 64 by applying 64 pointwise kernels with dimensions $1 \times 1 \times 384$. This is shown in Figure 1c.

Because the feature detection in the bottleneck layers is depthwise, the MAC count scales only linearly with the number of intermediate channels (384), instead of almost quadratically in a full convolution. Furthermore, the bottleneck structure reduces the cost of combining inter-channel information: in the example from Figure 1, this translates into $8^2 \times 384 \times (64+9+64) = 3,366,912$ MAC operations, many fewer than the $8^2 \times 9 \times 384^2 = 84,934,656$ operations that would have been needed for a full convolution.

## 2.2 CNN Accelerator Architecture

Specialized accelerators have been proposed to exploit the deterministic execution of CNNs and reduce their operational cost [9, 10, 19, 31, 40]. Figure 3 shows a typical 2D-array CNN accelerator and the associated processing element (PE) microarchitecture. A large on-chip global buffer (GLB) implemented in SRAM, together with smaller per-PE register files, allows for temporal reuse of data fetched from off-chip DRAM, amortizing the cost of DRAM accesses. DRAM and GLB reads can also be reused spatially by broadcasting them to different PEs simultaneously. Inside each PE, a MAC unit performs the computation required for the convolution operation, while register files allow for intra-PE temporal reuse.

In this paper, we use this accelerator architecture as a baseline, and modify it to efficiently support fused CNNs.

# 3 PREDICTIVE LAYER FUSION

Fusing two convolutional layers would be trivial if there were no non-linearity between them: the result would then be a linear composition of the two convolution operations. In reality, however, ReLU nonlinearities that separate the layers make this difficult.

Our key insight in resolving this conundrum is to *separate* the ReLU nonlinearity from the otherwise linear convolutional layers. We treat the ReLU layer as a *bitmask* layer where a cleared bit means that the corresponding activation will be zeroed while a set bit means that the activation will survive. If this bitmask is known ahead of time, the fused convolution is a matter of computing the combined weights that would result from a linear layer composition and applying them to the incoming activations.

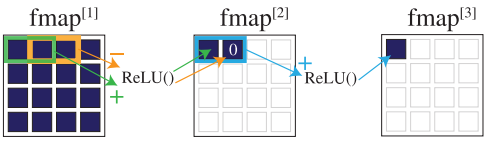



**Figure 4: Two consecutive convolution and ReLU layers. The output of second convolution in the first layer (shown by orange) is negative and replaced with zero after passing through ReLU.**

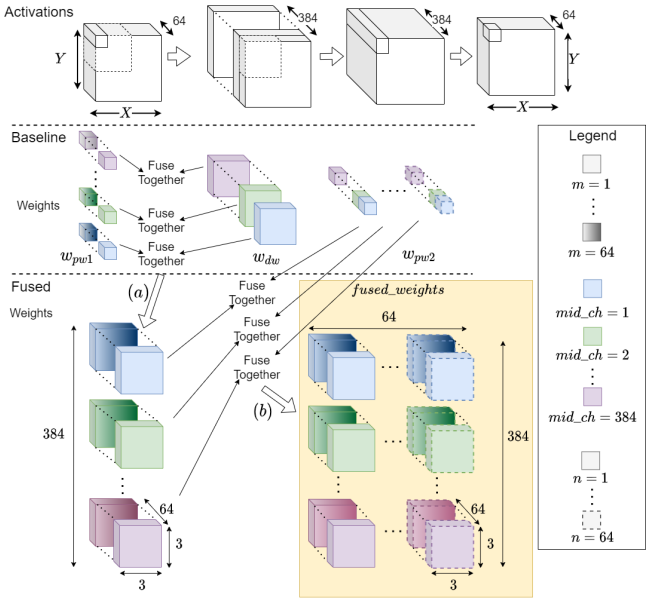



**Figure 5: Fusion of a MobileNet v2 bottleneck block. Different channels of PW1 are distinguished by fill (e.g., solid, gradient etc.), different PW2 filters are distinguished by border style, and different *mid_ch* are distinguished by colour. First, the PW1 ($w_{pw1}$) DW layers ($w_{dw}$) are fused (a), and then the PW2 layer ($w_{pw2}$) is fused with the result (b) to form the *fused_weights*.**

Below, we first describe how layers would be fused in the absence of ReLU. We then imagine that the ReLU bitmask is known through an oracle, and describe the layer fusion under these conditions. Finally, we describe how the oracle can be approximated by an on-line predictor.

## 3.1 Layer fusion without ReLU

Figure 4 illustrates two $2 \times 1$ convolutional layers with ReLU layer following each of them. In this section, we ignore the ReLU nonlinearities and treat the layers as if they were linear. The activation value in the fmap$^{[3]}$ coordinate $(1,1)$ is a function of the activations at $(1,1)$ and $(2,1)$ in the fmap$^{[2]}$, and those are functions of the activations at $(1,1)$, $(2,1)$, and $(3,1)$ in fmap$^{[1]}$. If we ignore ReLU, the output of the second layer can be computed directly from fmap$^{[1]}$:

$$
\begin{aligned}
fmap^{[3]}[1,1] &= fmap^{[1]}[1,1] \times \left(w^{[1]}[1] \times w^{[2]}[1]\right) \\
&+ fmap^{[1]}[2,1] \times \left(w^{[1]}[2] \times w^{[2]}[1] + w^{[1]}[1] \times w^{[2]}[2]\right) \quad (1) \\
&+ fmap^{[1]}[3,1] \times \left(w^{[1]}[2] \times w^{[2]}[2]\right)
\end{aligned}
$$

Since the weights are known prior to inference, all of the multiplications and additions that involve only weights in Equation (1) can be done *offline*; during inference, each activation is only multiplied by the resulting *reconstructed weight*. With fusion, the middle layer is effectively skipped, reducing the number of MACs required.

Figure 5 shows how the weights are fused for the more complex and general case of a MobileNet v2 bottleneck block. Fusing bottlenecks results in skipping the multiplications between the larger *mid_ch*-channel activation tensors. The layers are combined in two steps: (a) fusing the PW1 layer with the DW layer, and (b) fusing the result with the PW2 layer.

In step (a), each of the PW1 kernel filters — there are $mid\_ch = 384$ in total — is fused with the corresponding channel of the DW kernel. Since the filters are of different sizes ($1\times1$ versus $3\times3$), each weight in the $1 \times 1 \times 64$ PW1 kernel is multiplied with each of the nine weights of the corresponding DW kernel. Overall, this yields a fused layer with $3 \times 3 \times 64 \times 384$ filters which are the products of those from the PW1 and DW layers.

Then, in step (b), we fuse these resulting kernels with the 64 final PW2 kernels, each of which has $1 \times 1 \times 384$ weights. Each $1 \times 1$ filter is multiplied by $3 \times 3 \times 64$ weights of the corresponding channel produced in step (a). Doing this for all the 2 kernels yields a $3 \times 3 \times 64 \times 384 \times 64$ *fused_weights* tensor.

Using *fused_weights*, the final output fmap activations can be calculated as:

$$
\begin{aligned}
ofmap[x,y,n] = \sum_{kx=1}^{3}\sum_{ky=1}^{3}\sum_{m=1}^{M} (&ifmap[x+kx, y+ky, m] \\
&\times \sum_{i=1}^{mid\_ch} fused\_weights[kx, ky, m, i, n])
\end{aligned} \quad (2)
$$

where, in our running example, there are $n = 2$ bottleneck input channels, $mid\_ch = 384$ intermediate channels, and $m = 2$ bottleneck output channels.

The inner summation over *mid_ch* in Equation (2) is where the weights that need to be applied to the input feature map elements are reconstructed from combinations of *fused_weights* elements. The multiplicative and additive factors of inference-time batch normalization layers can be similarly folded into the fused weights.

Observe that this now requires *no multiplications*, only additions — that is, we have drastically reduced the number of multiplications by converting them to additions. This is the key source of efficiency in our proposal.

However, the *fused_weights* tensor has many more parameters than the total number of original weights for the bottleneck block — $3 \times 3 \times 64 \times 384 \times 64 = 14,155,776$ elements, much larger than $1\times1\times64\times384+3\times3\times384+1\times1\times384\times64 = 52,608$ original weights. We show how to overcome this new challenge in Section 4.2.

## 3.2 ReLU as a mask filter

Let us now consider how to fuse layers change if ReLU remains, but the effect of ReLU on each activation — in other words, the sign of each activation — is known through an oracle.

In the $2 \times 1$ convolution example from Figure 4, suppose output of the second (orange) convolution applied to $fmap^{[0]}$ is negative — denoted by a negative sign in the figure — and therefore replaced with a zero by ReLU. Correspondingly, in Equation (1), $w^{[2]}[2]$ becomes zero, which also zeroes any *fused_weights* terms that involve $w^{[2]}[2]$:

$$
\begin{aligned}
fmap^{[3]}[1,1] &= fmap^{[1]}[1,1] \times \underbrace{(w^{[1]}[1] \times w^{[2]}[1])}_{fused\_weights[1][1]} \\
&+ fmap^{[1]}[2,1] \times (\underbrace{w^{[1]}[2] \times w^{[2]}[1]}_{fused\_weights[2][1]} + \underbrace{w^{[1]}[1] \times \cancel{w^{[2]}[2]}}_{fused\_weights[1][2]}{}^{0}) \quad (3) \\
&+ fmap^{[1]}[3,1] \times (\underbrace{w^{[1]}[2] \times \cancel{w^{[2]}[2]}}_{fused\_weights[2][2]}{}^{0})
\end{aligned}
$$

In effect, ReLU can be thought of as a *bit-mask filter* that decides which pre-computed fused weights contribute to the final weight applied to each input feature map activation. For example, in a bottleneck block of MobileNet v2, where the output of the ReLU layer is $8^2 \times 384 = 24,576$ activations, the corresponding ReLU mask will have 24,576 bits.

As each bottleneck block has two ReLU layers, there will be two ReLU mask layers, which we will call $R1$ and $R2$. Rewriting Equation (2) to use $R1$ and $R2$, we obtain a general equation for a fused bottleneck block:

$$
\begin{aligned}
ofmap[x,y,n] = \sum_{kx=1}^{3}\sum_{ky=1}^{3}\sum_{m=1}^{M} (&ifmap[x+kx, y+ky, m] \\
&\times \sum_{i=1}^{mid\_ch} R1[x+kx, y+ky, i] \wedge R2[x,y,i] \\
&\times fused\_weights[kx, ky, m, i, n])
\end{aligned} \quad (4)
$$

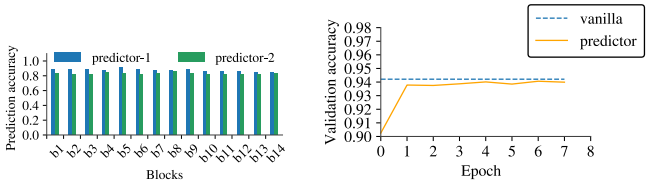
Because $R1$ and $R2$ entries are bits, their product can be replaced with a logical AND operation; pre-computed fused weights that survive the $R1 \wedge R2$ filter are then added together to produce the output feature map.

In the next section, we tackle the remaining task of determining the $R1$ and $R2$ masks.

## 3.3 Predicting the ReLU mask

To predict the mask bits at each position in the $R1$ and $R2$ ReLU filters, recall that this bit will be zero if and only if the corresponding pre-ReLU activation in the original (unfused) network would have been zero. In other words, each bit in $R1$ and $R2$ is the logical NOT of the *sign* bit of its activation in the original model.

Several prior works [7, 35, 36] predict the sign bit by using bit-serial arithmetic and processing the most significant bits first; this partial sum can then be used to decide whether the rest of the computation is necessary, since negative outputs will be zeroed by ReLU.




**(a) layer wise accuracy** **(b) overall accuracy**

**Figure 6: The occuracy of our ReLU predictor.**

While this can save computation time for single convolutions, it is not suitable for our use case because we need the mask reconstruct weights before any MAC operations are performed.

Our strategy for predicting $R1$ and $R2$ relies on the observation that only the sign bit needs to be predicted. We evaluate the original network *approximately*, with only enough precision that the sign bit can be reliably predicted (specifically, we use ternary $\{-1, 0, +1\}$ weights and 4-bit activations). The prediction step — evaluating the original model with ternary weights — turns all but one MAC operations into additions. At the required 4-bit width, this is far faster (and energetically cheaper) than evaluating the same layers at 16-bit precision.

Specifically, our predictor is based on TWN [23], which approximates the original weights as

$$W \approx \alpha \times W^t \tag{5}$$

where $W^t$ represents the ternary weights and $\alpha$ is a scalar chosen off-line to minimize the $\ell_2$ distance between the original weights and their approximations. In our case, we only need the *sign* of the final activation and not the magnitude, so $\alpha$ can be dropped for the depthwise layer prediction, while it can be merged with batch normalization for the pointwise layer.

Figure 6a shows how accurate the ReLU predictor is for $R1$ and $R2$. The predictor for $R2$ is always slightly less accurate than for $R1$, since it adds its own prediction error to the error carried through from $R1$. Nevertheless, all the predictors are more than 80% accurate.

It turns out that this level of accuracy is usually enough for the fused network to maintain the accuracy of the original unfused model. Figure 6b shows the CIFAR10 [21] classification accuracy for a variant of MobileNet v2 (see Table 1 for details) when inference is executed on the fused network (Equation (4)) and the ReLU masks are predicted using the TWN predictor. Initially, accuracy drops by about 2.5%, but after retraining the original accuracy is recovered.

## 4 FUSION-OPTIMIZED CNNS

Fusing bottleneck blocks significantly changes the tradeoff landscape of the original networks. On the one hand, because fusion removes the intermediate activations in each bottleneck, the number of intermediate channels (*mid_ch*) — and therefore the number of features detected — can be significantly increased in the pre-fused network. On the other hand, because fused blocks pre-compute weights for many combinations of the original network's weights, they need to store more weights than unfused networks.

**Table 1: Bottleneck blocks of original vs. fusion-optimized MobileNet v2 for CIFAR10. Each block is repeated n times, all with the same number of output channels. Stride is applied only in the first block of each sequence. Note the larger *mid_ch* and smaller input and output channels in the modified version.**

| input | *mid_ch* | *N* | *n* | stride | MACs (– / + fusion) |
|---|---|---|---|---|---|
| $32^2 \times 16$ | 144 | 24 | 2 | 1 | 13M / 9M |
| $32^2 \times 24$ | 192 | 32 | 3 | 1 | 38M / 26M |
| $16^2 \times 32$ | 384 | 64 | 4 | 2 | 50M / 33M |
| $8^2 \times 64$ | 576 | 96 | 3 | 1 | 76M / 56M |
| $8^2 \times 96$ | 960 | 160 | 3 | 2 | 61M / 38M |
| $8^2 \times 160$ | 960 | 320 | 1 | 1 | 30M / 29M |

**(a) original MobileNet v2**

| input | *mid_ch* | *N* | *n* | stride | MACs (– / + fusion) |
|---|---|---|---|---|---|
| $32^2 \times 16$ | 864 | 24 | 3 | 1 | 144M / 14M |
| $32^2 \times 24$ | 864 | 32 | 4 | 2 | 79M / 9M |
| $16^2 \times 32$ | 864 | 40 | 6 | 2 | 34M / 5M |
| $8^2 \times 40$ | 864 | 80 | 1 | 1 | 7M / 2M |

**(b) modified MobileNet v2**

In this section, we show how to adjust the DNN model architecture take advantage of the opportunities — and mitigate the challenges — presented by fusion.

### 4.1 Network architecture adaptation for fusion

In this section, we develop FusioNet, a bottleneck-based CNN optimized for fused bottlenecks. FusioNet outperforms MobileNet v2, Shufflenet, and Mnasnet in accuracy while requiring less computation after fusion.

**Rebalancing input/output and intermediate channels.** First, we consider rebalancing bottleneck blocks by increasing the number of intermediate channels and decreasing the number of channels at the bottleneck input and output — i.e., increasing *mid_ch* while decreasing $M$ and $N$ in Figure 1 — while keeping the overall number of MACs in the original unfused network roughly constant.

For example, to process a block with 32 input and output channels, 192 *mid_ch* and $8^2$ fmaps, we need $8^2 \times 192 \times (32 + 9 + 32) = 897,024$ MACs (without fusion), and if we change the size of input and output channels to 16 while increasing *mid_ch* to 384, $8^2 \times 384 \times (16 + 9 + 16) = 1,007,616$ MACs are needed to process the block (also without fusion). With fusion, however, the modified block needs only a quarter of the MACs needed by the original block, at the expense of more additions.

Table 1 shows how this technique can be applied to MobileNet v2 [32] bottleneck blocks. To keep the overall number of operations similar, the modified CNN has 14 blocks and 264M MACs in total (unfused), while the original CNN has 16 blocks and 246M MACs (unfused). Our evaluation (Section 6.2) shows that this transformation retains the accuracy of the original network.

**FusioNet.** To design FusioNet, we combine this approach with group convolutions and shuffle units as employed by ShuffleNet [42]. We use 2,368 and 3,456 intermediate channels (see Table 2), but the

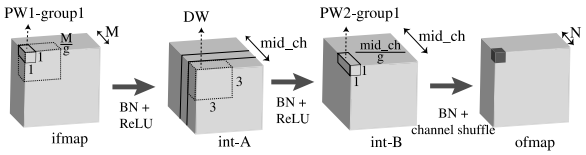



**Figure 7: FusioNet block structure. Residual connection between input and output of the block is not shown. Group convolution is performed with *g* groups.**

**Table 2: FusioNet: Each block is repeated *n* times, all with the same number of output channels. The stride is applied only in the first block of each sequence. Blocks are based on the structure shown in Figure 7 and *g* is the number of groups.**

| input | *mid_ch* | *N* | *n* | stride | g | MACs (– / + fusion) |
|---|---|---|---|---|---|---|
| $32^2 \times 32$ | 2368 | 32 | 3 | 1 | 4 | 182M / 7M |
| $16^2 \times 32$ | 2368 | 64 | 4 | 2 | 4 | 109M / 8M |
| $8^2 \times 64$ | 3456 | 96 | 8 | 2 | 4 | 103M / 10M |
| $8^2 \times 96$ | 3456 | 160 | 1 | 1 | 4 | 16M / 2M |

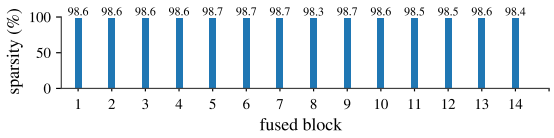



**Figure 8: Sparsity of fused bottleneck blocks for the CNN in Table 1.**

pointwise convolutions at each step are performed in four *groups*: that is, blocks with *mid_ch* = 3456 can be thought of as four separate blocks with *mid_ch*/4 channels, produced by separate PW1 layers that operate on 1/4 of the input channels. As with ShuffleNet, the channels are shuffled after the PW2 layer.

Table 2 details the resulting network architecture. While the number of *unfused* MACs is higher than the networks in Section 4.1, most of this is on the intermediate blocks; as a result, FusioNet actually becomes *more* efficient once bottleneck fusion is applied. Our evaluation (Section 6.3) shows that FusioNet can be executed much more efficiently than other CNNs with similar accuracy.

## 4.2 Pruning

Because fusing bottleneck layers requires computing fused weights that correspond to *combinations* of the original weights (see Section 3), at first sight it would appear that the total number of weights that must be stored and retrieved from off-chip memory would grow dramatically.

To ameliorate this problem, we first apply weight pruning [14, 15] to create sparsity in the original (unfused) weights. With retraining, this does not sacrifice accuracy [14, 15], and sparse models can be executed efficiently on a range of accelerator designs [3, 12, 13, 31].

Next, we observe that sparsity is *dramatically higher* for fused weights than for unfused weights. This is because a zero-valued weight can make *multiple* non-zero weights in the next layer ineffectual when multiplied with them during layer fusion. For example, if pointwise layers of a bottleneck are 80% sparse and the depthwise layer is 70% sparse, the *fused_weight* tensor for that block can achieve up to $1 - (0.2 \times 0.2 \times 0.3) = 98.8\%$ sparsity, depending on the exact distribution of zero weights in the unfused weight tensor.

Figure 8 shows the actual sparsity for the fused weights of MobileNet v2 in Table 1 when pruning is applied at 80% for the pointwise layers and 70% for the depthwise layers: the fused weights are on average 98.6% sparse.

Throughout the rest of the paper, we apply pruning to both the fused models and the unfused baseline models. In the next section, we describe an accelerator architecture that efficiently reconstructs the per-activation effective weights at runtime from the ultra-sparse fused weights tensor.

# 5 ARCHON ARCHITECTURE

Unfortunately, fused-layer inference is not efficient on existing DNN inference accelerators (e.g., Figure 3) because it uses many more adders than multipliers — additions are needed both to compute the effective activation weights from the fused weights that survive the ReLU mask (Section 3.1), and replace multiplications in the ternary-weight prediction phase (Section 3.3). Another challenge is the extreme (> 98%) sparsity present in the *fused_weights* structure (Section 4.2), together with the need to ensure sufficient throughput from the adders that reconstitute weights before applying them to activations. Finally, operations occur at different bitwidths: 4 bits during the prediction phase, and more (in our case 16) bits during the fused inference phase.

Below, we develop Archon, an accelerator architecture suitable for fusion. Like many prior accelerators [9, 19], Archon is a 2D array of processing elements (PEs); however, Archon PEs have significant microarchitectural changes to address the challenges outlined above.

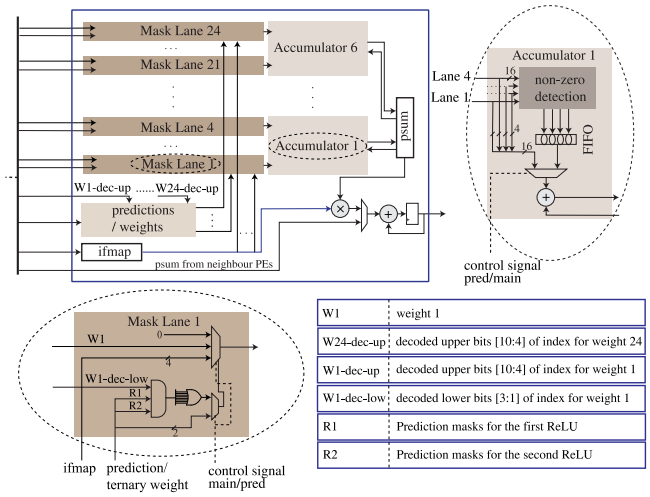
## 5.1 Data Reuse and Dataflow

**Convolutional Weight Reuse.** Inference with fused layers preserves convolutional weight reuse across inputs, as the *fused_weights* tensor does not depend on the input fmap $X$ or $Y$ dimensions. The weights are reused as in regular convolutions — that is, we can reuse a $3 \times 3 \times$ *mid_ch* weight block (for a $3 \times 3$ filter), rather than reusing a single weight (in pointwise layers) or a $3 \times 3$ filter (in depthwise layers) only within each channel. During the ReLU prediction phase, reuse is the same as with the original bottlenecks.

**ReLU Mask Reuse.** In addition to weights, the ReLU prediction masks can also be reused. Since the ReLU bitmasks $R1$ and $R2$ are independent of the input and output channels $M$ and $N$, they can be fetched once and reused spatially across multiple PEs that work on different input and output channels. The fetched masks $R1$ and $R2$ can also be reused temporally within each PE if the PE works on different bottleneck-input and bottleneck-output channels across different cycles. Finally, the $R2$ bitmask is reused across the filter size (e.g., $3 \times 3$) from the preceding convolutional layer.

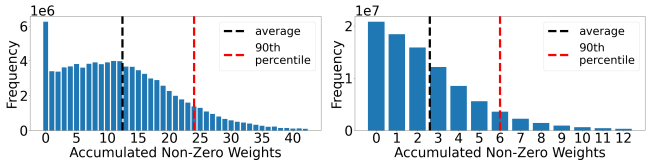
**Dataflow.** To maximize reuse opportunities, Archon uses different dataflows in the ReLU prediction phase and in the fused layer evaluation phase.

In the ReLU prediction phase, weight reuse is the same as the original bottleneck blocks, and limited in comparison to the fused




| | |
|---|---|
| W1 | weight 1 |
| W24-dec-up | decoded upper bits [10:4] of index for weight 24 |
| W1-dec-up | decoded upper bits [10:4] of index for weight 1 |
| W1-dec-low | decoded lower bits [3:1] of index for weight 1 |
| R1 | Prediction masks for the first ReLU |
| R2 | Prediction masks for the second ReLU |

**Figure 9: Archon PE microarchitecture. Colored components are modified or extra components needed compared to a conventional PE accelerator such as work [9]. Each PE has 6 accumulator modules and 4 identical lanes per accumulator. Accumulator and Lane 1 are shown in details. A reference table is provided that describes the data passed between PE components.**




**(a) before applying ReLU** **(b) after applying ReLU**

**Figure 10: Histogram of number of non-zero fused weights to be accumulated in weight reconstruction before and after applying ReLU during FusioNet inference on CIFAR100.**

evaluation stage. We therefore employ a dataflow that balances weight and activation reuse: (i) we distribute one of the input dimensions ($X$ in figure 5) horizontally across the PE array, and the output channels of each convolution (either *mid_ch* or $N$) across the other PE array dimension; and (ii) we further map the input channel dimension temporally within each PEs, so that the ternary convolution can be done locally inside the PEs.

During fused-layer evaluation phase, we focus on the opportunities for weight reuse: we use a weight-stationary dataflow that distributes the $X$ and $Y$ dimensions of the input fmaps across the horizontal and vertical dimensions of the PE array.

## 5.2 Data Representation

**Fused Weights.** Fused weights are compressed to take advantage of sparsity. Each individual segment along the *mid_ch* dimension is compressed separately in COO (coordinate) format [6], also used in other accelerator designs (e.g., [3, 22, 28]). In COO, each non-zero weight is stored as a (*value*, *offset*) pair, where the offset indicates the weight's index along the *mid_ch* dimension. We allocate 10 bits per offset, which allows for *mid_ch* to be as large as 1024.

**Ternary Weights.** The quantized weights used for ReLU prediction (discussed in section 3.3) are represented and stored densely as 2-bit values.

**Activations.** Activations are stored without compression.

## 5.3 PE microarchitecture

Like most DNN accelerators [9, 10, 19, 31], Archon is a 2D array of processing elements similar to figure 3, supported by a shared, banked global buffer (GLB). The Archon PEs, however, differ significantly from the simpler MAC-and-registers PEs because they need to efficiently support two features: (a) applying the ReLU filter masks ($R1$ and $R2$) to incoming fused weights, and (b) accumulating the fused weights that survived the ReLU filters.

Figure 9 shows the microarchitecture of a single Archon processing element. It comprises three main components: *masking lanes*, *accumulator* units, and the MAC unit. In addition, Archon PEs also needs a buffer to store the ReLU mask predictions $R1$ and $R2$, in addition to the weight, ifmap, and ofmap buffers present in the baseline PE from Figure 3.

Because the PEs are larger than baseline PEs, Archon has fewer PEs in the same silicon area than the baseline unfused accelerator. Nevertheless, the parallelism in the mask lanes and accumulator units, together with the far fewer MAC operations needed, means that Archon has higher throughput than the baseline unfused accelerator (see Section 6.3).

**Masking lanes.** During the fused layer evaluation stage, the masking lanes apply the predicted ReLU masks to fused weights. During the ReLU prediction phase, they are used to filter out activations for which the ternary weights are 0. They therefore consist of a mux that selects from among 0, the weight, and the 4-bit activation, and a unit that selects two bits (either $R1$ and $R2$ or the ternary weight) that correspond to the 16-bit weight or 4-bit activation from a 16-bit word. The selected bits are used to control the mux that zeroes the weight or activation.

**Accumulator units.** During fused evaluation, the accumulator units add together those fused weights that were not set to 0 in the masking lanes. During the ReLU prediction phase, they perform the ternary-weight convolution by either adding or subtracting the activations as dictated by the corresponding ternary weight. In prediction mode, each 16-bit adder operates as four 4-bit adders via gating the carry chains; this allows 24 partial sums for the ternary convolutions to be produced in each cycle.

The accumulator units first detect whether each incoming weight (or activation) is zero; the non-zero elements are buffered in a short first-in-first-out (FIFO) structure. As the non-zero elements leave the FIFO, they are added together, and sent on to the MAC unit whenever a full accumulated weight (or, in the prediction phase, convolution output) has been accumulated. The FIFO allows the accumulator to operate at line rate in the presence of irregular weight and activation sparsity.

To select the number of logic lanes per pipeline, and the number of pipelines per PE, we profiled several inference runs with FusioNet on CIFAR100. We collected the number of non-zero weights in each *mid_ch* after pruning, as well as the number of accumulated

weights after applying ReLU prediction; the results are shown in Figures 10a and 10b, respectively. 90% of the time, fewer than 24 non-zero fused weights are fetched in each *mid_ch*, and at most 6 weights are ultimately accumulated after ReLU prediction. Thus, to ensure full throughput, we employ six accumulators in each PE, and four masking lanes for each accumulator (24 lanes in total).

With this design, prediction masks can be reused among all six accumulators in each PE, provided that the pipelines receive weights at coordinates that differ only in the input and output channel IDs ($M$ and $N$ in Figure 1).

### 5.4 Operation Walk-Through

Archon operates in two different phases for each bottleneck block: the prediction of the ReLU mask bits, illustrated with a walk-through example in Figure 11, and the execution of the fused layers to compute the final results, shown in Figure 12. We walk through each phase below.

**Prediction phase.** Figure 11 shows how ReLU mask prediction operates for a pointwise convolution on a $4 \times 4$ feature map with 2 channels ❶; the output feature map (i.e., the intermediate-channel activations) has 4 channels (i.e., *mid_ch*=4) ❷. In the walkthrough example, these are distributed spatially among four PEs, with the *mid_ch* channels distributed horizontally and the $X$ dimension of the ifmap distributed vertically ❸. For clarity, we only show one of the PEs, and use one adder ❹ and two logic lanes ❻ in the PE.

In **step 0**, ternary weights and 4-bit ifmaps are fetched for the worktile in each PE and stored in the weight and ifmap buffers.

In **step 1**, the activation at the head of the buffer is broadcast to both lanes. Each lane also receives different ternary weights that correspond to the same activation but different intermediate channels. The LSB of each ternary weight is used to control the mask lane mux ❻ to zero activations if the ternary weight is zero.

In **step 2**, the higher bit the ternary weight in each lane is used to decide whether the activation should be accumulated or subtracted from the partial sum being accumulated ❼. In prediction mode, the adder operates as one separate 4-bit adder for each lane.

In **step 3**, the partial sums are either fed back into the adders or written back to the ofmap buffer ❽. Finally, the sign is extracted to determine the corresponding ReLU mask bit, and the activation is stored in the memory hierarchy to be read in the next layer.

Finally, **step 4** occurs when a partial sum is accumulated over all the input channels, and applies batch normalization (BN) to the activations. For the first ReLU layer, this requires both the additive and multiplicative parts of BN because the activations are used later; for the second ReLU layer, only the additive component is needed because only the sign is needed to compute the ReLU mask.

Predicting the ReLU masks for the next (and final) ReLU layer goes through the same process, but only uses the activations to determine the corresponding mask bits, and does not store them.

**Fused layer convolution phase.** Figure 12 shows a walk-through example of the same hardware operating in the fused convolution phase. It shows two accumulator units ❽ with their corresponding non-zero detectors ❼, and two lanes ❺/❻ for each accumulator unit. Each accumulator operates on weights corresponding to a different intermediate channel in the original bottleneck (*mid_ch*).

In **step 0**, the fused weight indices are fetched for both accumulator units; for each accumulator, we fetch enough weights to fill all masking lanes for each accumulator (in this figure, 2 per accumulator unit). The indices are split into upper bits, which are used to address the ReLU prediction mask buffer, and lower bits, which are used to extract ReLU bits from the fetched mask word.

In **step 1**, the prediction masks are fetched from the prediction buffer at word granularity. In this example, the two MSBs of the weight address are decoded to a 4-bit one-hot bit vector ❸, and used to decide which words for $R1$ and $R2$ should be fetched from the prediction buffer ❷.In the example, based on the decoded indices, words 1 to 3 will be fetched.

In **step 2**, the LSBs of each weight index ❹ are sent to their masking lanes, where they are anded ❺ with the ReLU prediction words from **step 1** to select the relevant $R1$ and $R2$ mask bits and and them together. This results in a single bit ❻ that indicates whether the non-zero fused weight ❶ should be replaced by zero due to either of ReLU masks.

In **step 3**, the non-zero detection units ❼ receives 2 weights from the two lanes that feed the corresponding accumulator unit ❾, and discards the zero weights; non-zero weights are sent to the accumulator buffer ❽.

In **step 4**, the accumulator ❾ receives non-zero weights from the buffer and adds them to its running sum to reconstruct the weight that will eventually be applied to the corresponding activation. Completed reconstructed weights are written to the ofmap buffer.

Finally, in **step 5**, the reconstructed weights are read from the ofmap buffer and are multiplied by the related activations in the MAC unit ❿; this results in partial sums that are accumulated until the entire convolution operation is complete.

## 6 EVALUATION

### 6.1 Methods

**Architecture baselines.** We compare Archon to the state-of-the-art DNN accelerator in [9], as well as a version that supports inference with sparse weights but not bottleneck layer fusion. To obtain energy and latency, we use Timeloop [30] and Accelergy [39] to model the baseline accelerators, and a custom cycle-level model together with Accelergy to model Archon. The hardware accelerators were configured as shown in Table 3.

**DNN model baselines.** To evaluate FusioNet, we use three modern CNNs that rely on bottleneck blocks: MobileNet v2 [32], MnasNet [37], and ShuffleNet [42]. For training and accuracy measurements, all networks were implemented in TensorFlow [1].

**Task and training.** Models were trained using SGD with momentum on the CIFAR10 and CIFAR100 [21] image classification tasks for 146 epochs, composed of 6 warmup epochs with a linear LR schedule and 140 epochs with cosine decay LR scheduling [26]. Normalizaton, as well as random cropping and mirroring for augmentation, were used during training. For CIFAR100 experiments, all the networks had a PW convolution following their bottleneck blocks sequence that mapped to 1260 channels. For CIFAR10 experiments, FusioNet used the same mechanism while other networks had the same layers following their bottleneck sequence as in their original papers.

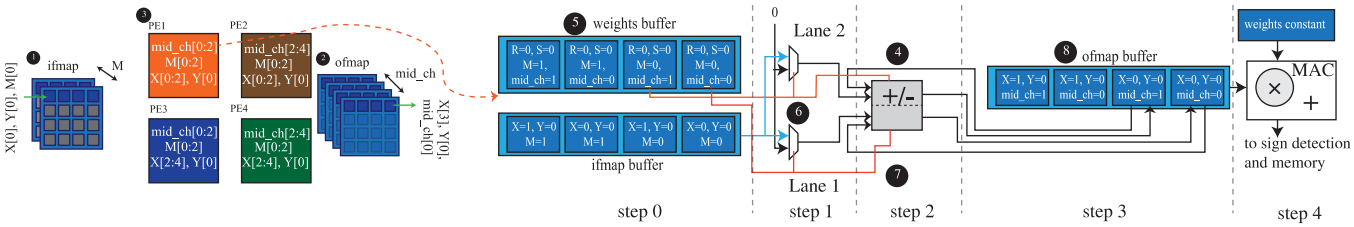



**Figure 11: Example of a ReLU prediction for a pointwise convolution in ARCHON. Spatial and temporal mapping of different dimensions are shown on the left and a snapshot of the first cycle in PE1 is shown on the right. Step0: filling buffers - Step1: applying ternary weight - Step2: ternary convolution - Step3: ofmap buffer access Step4: scale by constant**

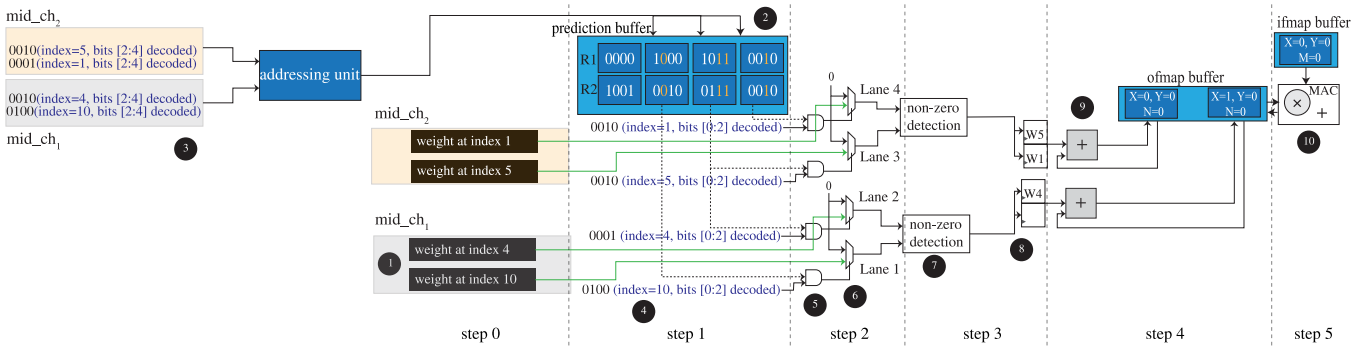



**Figure 12: Example of a fused computation for a bottleneck block in ARCHON PE. Step0: fetching and decoding weight indices - Step1: pred-buffer access - Step2: applying ReLU masks - Step3: filtering non-zero Step4: weight reconstruction - Step5: computing ofmap**

**Table 3: configurations of ARCHON and baseline accelerators**

| | ARCHON | baseline |
|---|---|---|
| Technology | 45nm | 45nm |
| Number of PEs | 64PEs | 72PEs |
| GLB | banks: 25<br>bank_size: 64 B | banks: 32<br>bank_size: 64 B |
| Buffer(per PE) | input(REGs): 64 B<br>output(REGs): 32 B<br>weight(SRAM): 256 B | input(REGs): 48 B<br>output(REGs): 32 B<br>weight(SRAM): 448 B |
| Compute(per PE) | MAC (16-bit): 1<br>Adders (16-bit): 6 | MAC (16-bit): 1 |
| Logic Lanes (per Adder) | 4 | N/A |
| Total Area | $1.53mm^2$ | $1.44mm^2$ |

**Table 4: CIFAR10 Validation accuracy after pruning and ReLU prediction. Models are (b) baseline or (m) modified as in Table 1.**

| CNN | vanilla | pruned | with pred. |
|---|---|---|---|
| MobileNet v2(b) | 94.26% | 93.45% | |
| MobileNet v2(m) | 94.27% | 94.01% | 93.5% |
| ShuffleNet(b) | 91.78% | 91.50% | |
| ShuffleNet(m) | 91.93% | 90.86% | 90.0% |
| MnasNet(b) | 94.53% | 94.16% | |
| MnasNet(m) | 94.39% | 93.64% | 93.03 % |
| FusioNet | 95.29% | 94.53% | 94.24% |

**Pruning.** We follow the approach in [43] and use bitmasks to deactivate lower-magnitude weights of a pretrained network, gradually increasing sparsity on a polynomial schedule. Once target sparsity is reached, fine-tuning recovers accuracy. We prune pointwise layers to 80% and depthwise layers to 70% sparsity.

**ReLU prediction.** Ternary weights used in the ReLU predictor are initialized using the pruned weights and quantized to ternary. They are then fine-tuned as in TWN [23], using the predicted ReLU mask as the activation function in the pruned network.

**Silicon area.** Area for memories and logic was estimated using CACTI [38] and Aladdin [33] through Timeloop [30]. Both storage and logic in ARCHON PEs were accounted for, and all accelerators were configured to take the same silicon area (see Table 3 for details).

## 6.2 Accuracy and Training Time

Tables 4 and 5 show the accuracy of FusioNet as well as the three baseline models for CIFAR-10 and CIFAR100 respectively, in the original form and in the version with bottlenecks resized for fusion as in Table 1. For each network, we also show the accuracy after the pruning and ReLU prediction steps.

All networks perform very close to the baseline accuracy, with only the modified ShuffleNet and modified MnasNet exceeding a 1%

**Table 5: CIFAR100 validation accuracy after pruning and ReLU prediction.**

| CNN | vanilla | pruned | with pred. |
|---|---|---|---|
| MobileNet v2 | 78.19% | 77.34% | |
| MnasNet | 78.16% | 77.05% | |
| FusioNet | 79.57% | 78.47% | 76.51% |

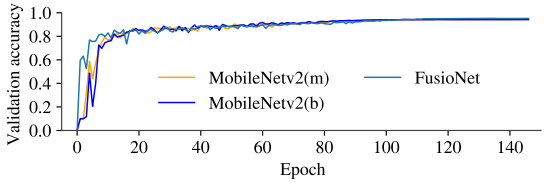
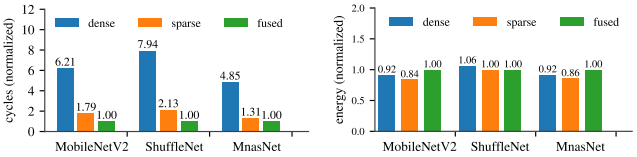



**Figure 13: Training curve for FusioNet, baseline and modified MobileNet v2 CNNs on the CIFAR10 dataset.**




**(a) total cycles** **(b) total energy**

**Figure 14: Inference runtime and energy for CIFAR10, normalized to the fused version. Dense and sparse networks are executed on baseline accelerators, and fused networks on Archon.**

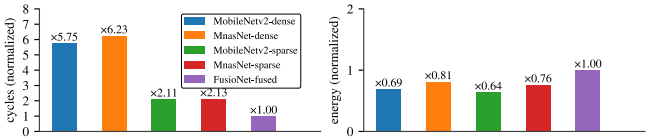
drop. FusioNet outperforms all of the baselines in the vanilla form — not unexpected as it has more MACs — but also matches the accuracy of the vanilla baselines on CIFAR10 even when pruning and ReLU prediction are applied. On CIFAR100, FusioNet experiences a slight accuracy drop when predictor is in place; nevertheless, accuracy is always within 1% of the pruned baseline models. FusioNet is also much more efficient when executed with layer fusion.

Figure 13 shows how accuracy improves when training FusioNet, the baseline and the modified MobileNet v2 models on CIFAR10. Both top accuracy and convergence time are very similar for all models, demonstrating that the modifications we use for efficient fusion do not affect training efficiency.

## 6.3 Inference Speedup and Energy

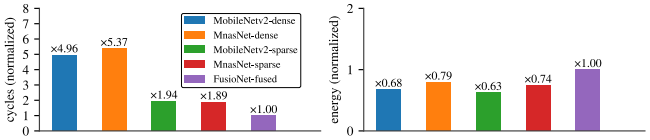
Figure 14a shows the execution times of the baseline CIFAR10 models — with and without weight sparsity — and fusion-optimized variants, executed on the baseline dense and sparse accelerators and Archon, respectively. The fusion-optimized models on Archon are much faster than the dense baseline (4.9×–8×) and noticeably faster than the sparse baseline (1.3×–2.1×).

Figure 15a, in turn, compares FusioNet on Archon to MobileNet v2 and MnasNet for CIFAR10 on the baseline accelerators. FusioNet is always faster, 5.8×–6.3× compared to the dense baseline and 2.1× compared to the sparse baseline. Similarly, Figure 16a shows the



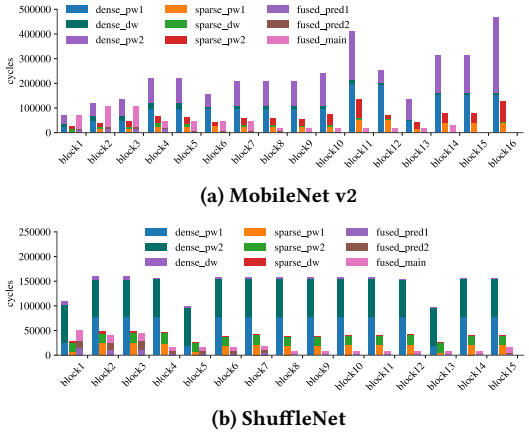

**(a) total cycles** **(b) total energy**

**Figure 15: CIFAR10 Inference runtime and energy. Data in both figures are normalized to the fusion execution of FusioNet. Dense and Sparse networks are executed on baseline accelerators while FusioNet on Archon.**




**(a) total cycles** **(b) total energy**

**Figure 16: CIFAR100 Inference runtime and energy. Data in both figures are normalized to the fusion execution of FusioNet. Dense and Sparse networks are executed on baseline accelerators while FusioNet on Archon.**




**(a) MobileNet v2**

**(b) ShuffleNet**

**Figure 17: Cycles breakdown for executing two CNNs for CIFAR10 with: (left) a dense accelerator, (middle) a sparse accelerator and (right) Archon.**

same comparison, but for CIFAR100. Like for CIFAR10, FusioNet is always faster, 4.9×–5.3× compared to the dense baseline and 1.9× compared to the sparse baseline. Note that the dimensions of the bottleneck blocks for the models in CIFAR10 and CIFAR100 are the same, and the main difference between the two networks is what follows the bottlenecks.

Finally, Figure 17 shows the breakdown of processing each bottleneck block of MobileNet v2 and ShuffleNet for CIFAR10 on the baseline accelerators and Archon. For the unfused networks, the


**(a) Executing fused FusioNet**




**(b) Fusion overhead for execution fused FusioNet**

**Figure 18: Energy breakdown for FusioNet for CIFAR10 on Archon.**

pointwise layers take up the lion's share of the cycles to map features to and from the intermediate channel dimension. This validates the key intuition behind layer fusion and Archon: removing the intermediate bottleneck component with many channel dimension through fusion. Fused networks on Archon are always faster; nearly all of the computation is in the fused convolution layers, with little overhead due to the ReLU prediction phase.

## 6.4 Energy

Figure 14b shows the overall energy consumption for the three CIFAR10 baselines, again in sparse, dense, and fused versions, and executed on the corresponding hardware; Figure 15b shows the same data when FusioNet is compared to the baselines. FusioNet needs around 34% more energy compared to iso-accuracy MobileNet and 26% more energy compared to iso-accuracy MnasNet, a cost that is much less than the improvement in inference speed. Figure 16b shows the same data but for CIFAR100 models, and the results are similar to the results for CIFAR10 models.

however, the problem is not severe and the speedup gained by FusioNet is still much more than the extra energy it needs.

Figure 18a shows the energy breakdown of executing FusioNet for CIFAR10 on Archon. The most significant sources of energy consumption are prediction steps for the two ReLU layers, followed by the extra operations needed to process the fused blocks (e.g., loading the fused weights and accessing the prediction buffer), denoted as fusion overhead. Most of the energy is spent on ReLU prediction; while this contributes little to execution time, it requires large intermediate-channel feature maps to be moved from and back to memory.

Finally, Figure 18b shows the breakdown of the overheads inherent in executing the fused convolutions for CIFAR10. Most of the overhead is due to streaming the fused weights from memory; this is both because there are more fused weights than original weights even after pruning, and the fact that Archon does not employ an on-PE weight buffer during fused execution. Next in line is accessing the prediction buffers, followed by the energy needed to store and refetch prediction masks from memory. Other operations such as accumulation and decoding the weight indices do not make a significant contribution to the overheads.

# 7 RELATED WORK

**Neural Network Accelerators.** A plethora of dense and sparse neural network accelerators have been proposed [3, 9, 10, 12, 13, 19, 22, 28, 31]. While each of these accelerators are designed for different types of sparsity, operation mappings, and distinct compression formats, none can efficiently execute neural networks that contain fused convolutions.

**Pruning.** In this work, we gradually prune our network using a magnitude-based, unstructured approach, as described by Zhu et al. [43]. However, the specific pruning method is orthogonal to our work, and other pruning techniques [24, 27, 41] can be used instead.

**Bottleneck Blocks.** The proposed FusioNet bears resemblance to many neural network designs [16, 17, 32, 37, 42] that utilize variants of bottleneck layers in order to make computations more efficient for edge devices. However, the dimensions of FusioNet are selected while keeping in mind layer fusion; specifically, fusion permits intermediate dimensions that would be impractically large in an unfused design, which results in increased accuracy.

**Prior Research on DNN "Fusion."** Some prior work has proposed very limited "fusion" within neural networks. Several of these proposed to fuse the convolutional layer with batch normalization and ReLU [5, 8, 20]. In contrast to our proposal, these transformations are mathematically straightforward, and do not fuse across nonlinearities. Likewise, the FusedCNN [4] accelerator also does not fuse the computation as we do in this paper; instead, layer computations are performed as normal while keeping the intermediate activations in on-chip storage. To the best of our knowledge, our work is the first design that replaces multiple layer computations with one fused computation.

**Low-Precision Evaluation as Prediction.** Several prior works have proposed using cheaper computations to predict potentially ineffectual computations in the future. Several [7, 35, 36] propose to perform bit-serial multiplications starting with the leading bits of the activations in order to predict the sign of the output and determine if the computation can terminiate early (since any negative outputs are eventually squashed to zero by ReLU). Similarly, SnaPEA [2] calculates the MACs that involve positive weights first, and only then decides whether the remaining MAC operations (with negative weights) are necessary. Other work [34] relies on depthwise convolutions to predict the signs of the main convolution output. Similarly, DUET [25] performs low-precision convolutions and uses them to predict which of the low-precision outputs require higher-precision arithmetic to keep the model accuracy the same as the baseline. These prior works use partial computation results only to terminate computation of single convolutions early; in contrast, our work uses a separate prediction computation to enable fusion across multiple convolutional layers.

# 8 CONCLUSION

This paper introduces (i) *predictive layer fusion* for efficient execution of CNN bottleneck blocks, (ii) a hardware architecture and dataflow designed to execute fused networks, and (iii) a technique to optimize CNN models for fusion.

Combined, these techniques yield up to 5.8 × faster inference compared to compact networks executed on a dense DNN accelerator, and 2.1 × faster inference compared to when these networks are pruned and executed on a sparse DNN accelerator.

## 9 ACKNOWLEDGEMENTS

The authors are grateful to the anonymous reviewers for insightful feedback and helpful suggestions.

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Project Agency (DARPA) under agreement number FA8650-20-2-7007, and by the Natural Sciences and Engineering Research Council of Canada (NSERC) under award number NETGP 485577-15. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL), Defense Advanced Research Project Agency (DARPA), the U.S. Government, the Natural Sciences and Engineering Research Council of Canada (NSERC), or the Government of Canada.

## REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafał Józefowicz, Łukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

[2] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K Gupta, and Hadi Esmaeilzadeh. 2018. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 662–673.

[3] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–13.

[4] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783725

[5] Michael Anderson, Evangelos Georganas, Sasikanth Avancha, and Alexander Heinecke. 2018. Tensorfolding: Improving convolutional neural network performance with fused microkernels. In *Proc. Int. Conf. High Perform. Comput., Netw., Storage, Anal.(SC)*.

[6] Brett W Bader and Tamara G Kolda. 2008. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing* 30, 1 (2008), 205–231.

[7] Jiho Chang, Yoonsung Choi, Taegyoung Lee, and Junhee Cho. 2018. Reducing MAC operation in convolutional neural network with sign prediction. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 177–182.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[9] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.

[10] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 92–104.

[11] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 807–820.

[12] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. 2019. SparTen: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 151–165.

[13] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*. 243–254.

[14] Song Han, Huizi Mao, and W. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2016).

[15] Song Han, Jeff Pool, John Tran, and William J Dally. 2015. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 1*. 1135–1143.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[17] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. 2019. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 1314–1324.

[18] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*. PMLR, 448–456.

[19] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 1–12.

[20] Wonkyung Jung, Daejin Jung, , Byeongho Kim, Sunjung Lee, Wonjong Rhee, and Jung Ho Ahn. 2019. Restructuring Batch Normalization to Accelerate CNN Training. arXiv:1807.01702 [cs.CV]

[21] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

[22] Ching-En Lee, Yakun Sophia Shao, Jie-Fang Zhang, Angshuman Parashar, Joel Emer, Stephen W Keckler, and Zhengya Zhang. [n.d.]. Stitch-x: An accelerator architecture for exploiting unstructured sparsity in deep neural networks.

[23] Fengfu Li and Bin Liu. 2016. Ternary Weight Networks. *CoRR* abs/1605.04711 (2016).

[24] Shaohui Lin, Rongrong Ji, Chenqian Yan, Baochang Zhang, Liujuan Cao, Qixiang Ye, Feiyue Huang, and David Doermann. 2019. Towards optimal structured cnn pruning via generative adversarial learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2790–2799.

[25] Liu Liu, Zheng Qu, Lei Deng, Fengbin Tu, Shuangchen Li, Xing Hu, Zhenyu Gu, Yufei Ding, and Yuan Xie. 2020. DUET: Boosting Deep Neural Network Efficiency on Dual-Module Architecture. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 738–750.

[26] Ilya Loshchilov and Frank Hutter. 2017. SGDR: Stochastic Gradient Descent with Warm Restarts. *arXiv: Learning* (2017).

[27] Sangkug Lym, Esha Choukse, Siavash Zangeneh, Wei Wen, Sujay Sanghavi, and Mattan Erez. 2019. PruneTrain: fast neural network training by dynamic sparse model reconfiguration. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[28] Bradley McDanel, Sai Qian Zhang, HT Kung, and Xin Dong. 2019. Full-stack optimization for accelerating cnns using powers-of-two weights with fpga validation. In *Proceedings of the ACM International Conference on Supercomputing*. 449–460.

[29] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*. 807–814.

[30] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. 2019. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 304–315.

[31] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 27–40. https://doi.org/10.1145/3079856.3080254

[32] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[33] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 97–108.

[34] Gil Shomron, Ron Banner, Moran Shkolnik, and Uri Weiser. 2020. Thanks for nothing: Predicting zero-valued activations with lightweight convolutional neural networks. In *European Conference on Computer Vision*. Springer, 234–250.

[35] Md Kamruzzaman Shuvo, David E Thompson, and Haibo Wang. 2020. MSB-First Distributed Arithmetic Circuit for Convolution Neural Network Computation. In *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*. 399–402. https://doi.org/10.1109/MWSCAS48704.2020.9184599

[36] Mingcong Song, Jiechen Zhao, Yang Hu, Jiaqi Zhang, and Tao Li. 2018. Prediction based execution on deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 752–763.

[37] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2820–2828.

[38] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P Jouppi. 2008. *CACTI 5.1*. Technical Report. Technical Report HPL-2008-20, HP Labs.

[39] Yannan N. Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8.

[40] Shunzhi Yang, Zheng Gong, Kai Ye, Yungen Wei, Zheng Huang, and Zhenhua Huang. 2019. EdgeCNN: Convolutional Neural Network Classification Model with small inputs for Edge Computing. arXiv:1909.13522 [cs.CV]

[41] Zhonghui You, Kun Yan, Jinmian Ye, Meng Ma, and Ping Wang. 2019. Gate decorator: Global filter pruning method for accelerating deep convolutional neural networks. *arXiv preprint arXiv:1909.08174* (2019).

[42] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 6848–6856.

[43] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. arXiv:1710.01878 [stat.ML]