Boosting the Throughput and Accelerator Utilization of Specialized CNN Inference Beyond Increasing Batch Size

Jack Kosaian [1 *] **Amar Phanishayee** [2] **Matthai Philipose** [2] **Debadeepta Dey** [2] **K. V. Rashmi** [1]

Abstract

Datacenter vision systems widely use small, specialized convolutional neural networks (CNNs) trained on specific tasks for high-throughput inference. These settings employ accelerators with massive computational capacity, but which specialized CNNs underutilize due to having low arithmetic intensity. This results in suboptimal application-level throughput and poor returns on accelerator investment. Increasing batch size is the only known way to increase both application-level throughput and accelerator utilization for inference, but yields diminishing returns; specialized CNNs poorly utilize accelerators even with large batch size. We propose *FoldedCNNs*, a new approach to CNN design that increases inference throughput and utilization beyond large batch size. FoldedCNNs rethink the structure of inputs and layers of specialized CNNs to boost arithmetic intensity: in FoldedCNNs, f images with C channels each are concatenated into a single input with fC channels and jointly classified by a wider CNN. Increased arithmetic intensity in FoldedCNNs increases the throughput and GPU utilization of specialized CNN inference by up to $2.5\times$ and $2.8\times$, with accuracy close to the original CNN in most cases.

1. Introduction

Convolutional neural networks (CNNs) are widely deployed for high-throughput vision tasks. Many such tasks target highly specific events for which general-purpose CNNs trained on diverse data (e.g., ResNet-50 on ImageNet) are overkill; an application detecting red trucks does not need a CNN capable of classifying animals. It has thus become popular to employ small *specialized CNNs* trained only for such focused tasks (Shen et al., 2017; Kang et al., 2017; Hsieh et al., 2018; Kang et al., 2020). In being trained for highly specific tasks, specialized CNNs can typically be much smaller than general-purpose CNNs, and thus operate at higher application-level throughput (e.g., images/sec).

Specialized CNNs are heavily used for inference in both datacenters and edge clusters (Kang et al., 2017; Hsieh et al., 2018; Mullapudi et al., 2019; Bhardwaj et al., 2020), and occasionally on constrained devices (e.g., cameras) (Canel et al., 2019). *We focus on specialized CNNs used for high-throughput vision in datacenters/clusters.* A popular usecase in this setting is offline video analytics, in which all video frames are processed by a specialized CNN, and only frames for which the specialized CNN is uncertain are processed by a slower, general-purpose CNN (Kang et al., 2017). The throughput of the specialized CNN is critical to that of the overall system, as all frames are processed by the specialized CNN and only a small fraction by the general-purpose CNN.

Aiding the case for high-throughput CNNs, server-grade deep learning hardware accelerators offer unprecedented performance in FLOPs/sec, and thus are used for inference in datacenters (e.g., V100 and T4 GPUs, TPUs) and edge clusters (e.g., AWS Outposts and Azure Stack Edge with T4 GPUs). It is critical that these accelerators be highly utilized, with software running on an accelerator ideally achieving FLOPs/sec near the accelerator's theoretical peak FLOPs/sec. Given the high cost of accelerators and the operational costs incurred in deploying them (e.g., power) (Barroso et al., 2013), poorly utilizing an accelerator leads to a poor return on investment. Furthermore, underutilization results in suboptimal application-level throughput.

However, current specialized CNNs significantly underutilize server-grade accelerators: we find that specialized CNNs used in production at Microsoft achieve less than 20% of the peak FLOPs/sec of GPUs employed in datacenters, even with large batch sizes (which are common for high-throughput inference), and when using techniques that improve throughput, such as reduced precision (see Fig. 1). While specialized CNNs might better utilize weaker devices, we find that server-grade GPUs, such as V100 and T4, offer the highest cost-normalized throughput for the CNNs described above, motivating their deployment in production.

[*]Work done in part as an intern at Microsoft Research. [1]Carnegie Mellon University [2]Microsoft Research. Correspondence to: Jack Kosaian <jkosaian@cs.cmu.edu>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

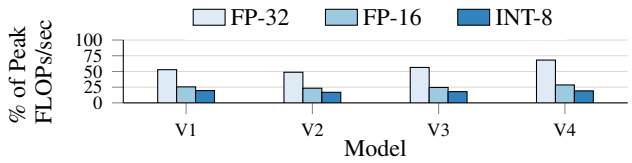


Figure 1. Utilization of production specialized CNNs (see §2.1) at various precisions and maximum batch size on a T4 GPU. Each bar is relative to the peak FLOPs/sec of the T4 in that precision.

The main cause for the poor accelerator utilization of specialized CNNs is low *arithmetic intensity*: the ratio between the number of arithmetic operations performed by a computational kernel (i.e., FLOPs) and the number of bytes read from or written to memory by the kernel (Williams et al., 2009). As the bandwidth of performing arithmetic on accelerators is far higher than memory bandwidth (e.g., over 200× on T4 (NVIDIA, 2018)), a CNN with low arithmetic intensity incurs frequent memory stalls, leaving arithmetic units idle and underutilized. High arithmetic intensity is, thus, a prerequisite to high utilization. However, we will show in §2 that specialized CNNs have arithmetic intensities far lower than needed for peak utilization on accelerators.

The arithmetic intensities of specialized CNNs must be increased to improve utilization of server-grade accelerators, but achieving this requires care: we show that common techniques that increase application-level throughput can reduce arithmetic intensity, while naive approaches to increasing arithmetic intensity reduce application-level throughput.

Increasing the batch size over which inference is performed *can* increase arithmetic intensity, utilization, and application-level throughput by amortizing the cost of loading a CNN's weights from memory. However, doing so leads to diminishing returns in these quantities: for example, we show in §2 that specialized CNNs achieve at most 17% of the peak FLOPs/sec of a V100 at large batch sizes. *An alternative is needed to further improve the utilization and throughput of specialized CNNs beyond the limits of increasing batch size.*

We propose *FoldedCNNs*, a new approach to the design of specialized CNNs that boosts inference utilization and throughput beyond increasing batch size. We show that convolutional and fully-connected layers in specialized CNNs at large batch size can be transformed to perform an equal number of FLOPs, but with higher arithmetic intensity. Our key insight is that, once arithmetic intensity has plateaued due to increased batch size, reading/writing activations accounts for most of the memory traffic in specialized CNNs. We show that this memory traffic can be significantly reduced, while performing the same number of FLOPs, by jointly decreasing the size of the batch of input/output activations for a layer and increasing the layer's width. By decreasing memory traffic while performing the same number of FLOPs, this transformation increases arithmetic intensity.

FoldedCNNs take a new approach to structuring the inputs

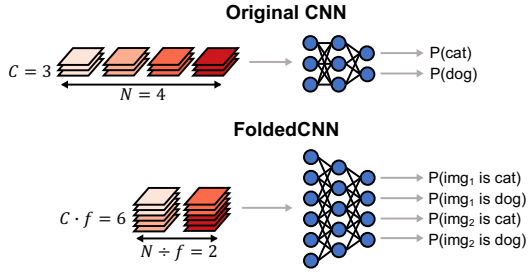


Figure 2. Abstract illustration of a FoldedCNN with $f = 2$.

of a CNN to apply this transformation, inspired in part from the interplay of machine learning and coding theory in other applications (Kosaian et al., 2020). As shown in Fig. 2, rather than operating over a batch of N images each with C channels, a FoldedCNN instead operates over a batch of $\frac{N}{f}$ "folded" inputs each with fC channels formed by concatenating f images along the channels dimension. These f images are jointly classified: if the original CNN had C_L output classes, the FoldedCNN now has fC_L output classes. FoldedCNNs increase the number of channels for all middle layers by $\sqrt{f}\times$, while maintaining an $f\times$ reduction in batch size. This reduces memory traffic over N images by $\sqrt{f}\times$ while performing a similar number of FLOPs, thus increasing arithmetic intensity, utilization, and throughput.

We evaluate FoldedCNNs on four specialized CNNs used at Microsoft and four from the NoScope video-processing system (Kang et al., 2017). FoldedCNNs improve the GPU utilization of specialized CNNs by up to 2.8× and throughput by up to 2.5×, while maintaining accuracy close to the original CNN in most cases. Compared to the compound scaling used in EfficientNets (Tan & Le, 2019), FoldedCNNs achieve higher accuracy, throughput, and utilization for specialized CNNs. These results show the promise of FoldedCNNs in increasing the utilization and throughput of specialized CNNs beyond increased batch size, and open doors for future high-performance specialized CNNs. The code used in this paper is available at `https://github.com/msr-fiddle/folded-cnns`.

2. Challenges in Achieving High Utilization

We now describe challenges in achieving high accelerator utilization in specialized CNN inference.

2.1. Specialized CNNs

As described in §1, specialized CNNs are small CNNs designed to target highly specific visual tasks and to achieve higher throughput than large, general-purpose CNNs. We focus on two motivating usecases of specialized CNNs:

Usecase 1: filters. A popular use of specialized CNNs is as lightweight filters in front of slower, general-purpose CNNs. In such systems, all video frames/images pass through a specialized CNN, and are processed by the general-purpose CNN only if the specialized CNN is uncertain (Kang et al.,

Table 1. Specialized CNNs used in this work.

Group	ID	Name	Resol.	Convs.	Classes
NoScope	N1	coral	(50, 50)	2	2
	N2	night	(50, 50)	2	2
	N3	roundabout	(50, 50)	4	2
	N4	taipei	(50, 50)	2	2
Microsoft	V1	lol-gold1	(22, 52)	5	11
	V2	apex-count	(19, 25)	5	22
	V3	sot-coin	(17, 40)	5	15
	V4	sot-time	(22, 30)	8	27

Table 2. Parameters of a 2D convolution with stride of 1.

Parameter(s)	Variable(s)
batch size	N
output height, width	H, W
input, output channels	C_i, C_o
conv. kernel height, width	K_H, K_W

2017). In other cases, the specialized CNN builds an approximate index to accelerate later ad-hoc queries by a general CNN (Hsieh et al., 2018). These applications desire high throughput, so batching is heavily exploited. We use specialized CNNs from the NoScope video-processing system (Kang et al., 2017) as examples of this usecase.

Usecase 2: game scraping. We also consider specialized CNNs used in production at Microsoft to classify events in video game streams by scraping in-game text appearing in frames (e.g., score). Separate CNNs are specialized for each game and event type. The service handles thousands of streams at once, and thus heavily batches images.

Comparison of general and specialized CNNs. General-purpose CNNs, such as those used for ImageNet, have many convolutional layers, each with many channels. For example, ResNet-50 has 49 convolutional layers, each with 64–2048 channels. In contrast, specialized CNNs have far fewer layers and channels: the specialized CNNs used in NoScope (Usecase 1) have 2–4 convolutional layers, each with 16–64 channels; those used at Microsoft (Usecase 2) have 5–8 convolutional layers with at most 32 channels. Further details on these CNNs are given in Table 1 and §A.

2.2. High utilization requires high arithmetic intensity

As described in §1, achieving high utilization of accelerators is critical for operational efficiency. Ideally, a CNN would operate near the peak FLOPs/sec offered by an accelerator. However, achieving this is confounded by the need to transfer data to/from memory, as cycles stalled on memory are wasted if they cannot be masked by computation.

A computational kernel must be compute bound to achieve peak FLOPs/sec: a compute-bound kernel uses all arithmetic units on an accelerator at all times. Under the popular Roofline performance model (Williams et al., 2009), a kernel can only be compute bound if it theoretically spends more time computing than it does reading/writing memory:

$$\frac{\text{FLOPs}}{\text{Compute Bandwidth}} > \frac{\text{Bytes}}{\text{Memory Bandwidth}}$$

Here, "FLOPs" is the number of arithmetic operations performed, "Bytes" is the amount of data transferred to/from memory (memory traffic), "Compute Bandwidth" is the accelerator's peak FLOPs/sec, and "Memory Bandwidth" is the accelerator's memory bandwidth (bytes/sec). Rearranging this to pair properties of the kernel on the left-hand side and properties of the accelerator on the right-hand gives:

$$\frac{\text{FLOPs}}{\text{Bytes}} > \frac{\text{Compute Bandwidth}}{\text{Memory Bandwidth}} \tag{1}$$

The left-hand ratio of Eqn. 1 is termed "arithmetic intensity": the ratio between the FLOPs performed by the kernel and the bytes it transfers to/from memory. The arithmetic intensity of a given layer in a CNN is (abstractly) written as:

$$\frac{\text{FLOPs}}{\text{Input bytes} + \text{Weight bytes} + \text{Output bytes}} \tag{2}$$

where "Input bytes" is the size of the layer's input activations, "Output bytes" is the size of output activations written by the layer to memory for processing by the next layer, and "Weight bytes" is the size of the layer's weights. For example, using the terminology in Table 2, the arithmetic intensity of a 2D convolutional layer with a stride of 1 is:

$$\frac{2NHWC_oC_iK_HK_W}{B(NHWC_i + C_iK_HK_WC_o + NHWC_o)} \tag{3}$$

where B is numerical precision in bytes (e.g., 2 for FP-16).[1] The aggregate arithmetic intensity of a CNN as a whole is computed by summing the FLOPs performed by each layer of the CNN, summing the bytes read/written by each layer, and dividing these quantities. This accounts for optimizations like layer fusion that reduce memory traffic.

Eqn. 1 indicates that, for a kernel to achieve the peak FLOPs/sec of an accelerator, the kernel's arithmetic intensity must be higher than the ratio between the accelerator's compute bandwidth and memory bandwidth (Williams et al., 2009).[2] For example, this ratio is 139 in half-precision on a V100 GPU (NVIDIA), 203 on a T4 GPU (NVIDIA, 2018), and 1350 on TPUv1 (Jouppi et al., 2017). It is often necessary for arithmetic intensity to be far higher than this ratio, as arithmetic intensity calculations typically assume perfect memory reuse, which can be difficult to achieve in practice.

Specialized CNNs have low arithmetic intensity. While high arithmetic intensity is needed for high utilization of

[1] Here, we show arithmetic intensity for direct- and GEMM-based convolutions, though the arguments we make also apply to other implementations (e.g., Winograd), as we show in §D.

[2] This condition is necessary, but not sufficient, as inefficiencies in implementation can limit performance (Williams et al., 2009).

accelerators, specialized CNNs have low arithmetic intensity due to their small sizes. For example, the half-precision arithmetic intensities of the CNNs used in the game-scraping tasks are 88–102 at large batch sizes, much lower than the minimum of 139 required for peak utilization of a V100 GPU, which is used for specialized CNN inference in datacenters (Mullapudi et al., 2019). Thus, these CNNs achieve at most 17% of the V100's peak FLOPs/sec, even at large batch sizes and when running on the TensorRT inference library that performs optimizations like layer fusion. To improve their utilization of accelerators, specialized CNNs must be modified to increase arithmetic intensity.

As described above, high arithmetic intensity alone is insufficient to achieve high utilization, as implementations must efficiently use accelerator resources (e.g., memory hierarchy). Nevertheless, high arithmetic intensity is a prerequisite for high utilization. For specialized CNNs, increasing arithmetic intensity is thus necessary to increase utilization. We will show that simply increasing arithmetic intensity greatly increases the utilization and throughput of specialized CNN inference atop an optimized inference library.

2.3. Improving arithmetic intensity is non-trivial

To increase the arithmetic intensity of convolutional and fully-connected layers, one must increase the ratio in Eqn. 2. For concreteness, we focus on convolutional layers in this subsection, and thus on increasing Eqn. 3.

Low precision? One way to increase Eqn. 3 is to decrease numerical precision B, which reduces memory traffic by representing operands/outputs using fewer bits. However, modern accelerators have compute units that offer increased FLOPs/sec in low precision (e.g., T4 GPUs). Reducing precision thus increases both the left-hand side of Eqn. 1 (by reducing bytes) and the right-hand side (by increasing compute bandwidth). When these quantities change at equal rates, as is common in accelerators (NVIDIA, 2018), the inequality remains the same: *kernels that did not satisfy this inequality at a high precision will not satisfy it at low precision.* Fig. 1 illustrates this on a T4 GPU: specialized CNNs have low utilization at both full (FP-32) and low precisions (FP-16, INT-8). Thus, while reducing precision can accelerate inference, it *does not* increase utilization.

Large batch size? Increasing batch size N can increase arithmetic intensity by amortizing the cost of loading layer weights. However, doing so leads to diminishing returns in arithmetic intensity (A), as (ignoring B in Eqn. 3):

$$A = \frac{2NHWC_oC_iK_HK_W}{NHWC_i + C_iK_HK_WC_o + NHWC_o}$$
$$\lim_{N\to\infty} A = \frac{2C_oC_iK_HK_W}{C_i + C_o} \quad (4)$$

When batch size is large enough that arithmetic intensity is determined by Eqn. 4, we say that a layer is in the "batch-limited regime." Fig. 3 shows this on the game-scraping CNNs: arithmetic intensity and utilization plateau with large batch size at 17% of the peak FLOPs/sec of a V100.

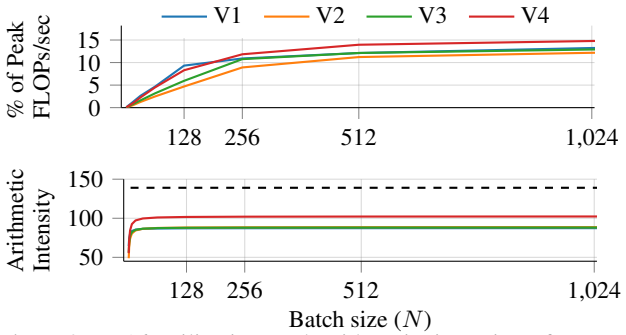


Figure 3. FP-16 utilization and arithmetic intensity of game-scraping CNNs on a V100 GPU. The dashed line is the minimum arithmetic intensity needed for peak utilization of a V100 GPU.

To further increase arithmetic intensity beyond the limits of increased batch size, Eqn. 4 indicates that one must increase C_i, C_o, K_H, or K_W. However, doing so increases the number of FLOPs performed by the layer per image, which typically *decreases application-level throughput*.

Takeaway. To increase utilization beyond increasing batch size, while maintaining high throughput, one must increase arithmetic intensity without greatly increasing FLOP count. We next propose techniques to achieve this goal.

3. Boosting Intensity via Folding

We now propose transformations to increase the arithmetic intensity of layers of specialized CNNs operating over large batches without increasing FLOPs. For clarity, we focus on convolutional layers, though the transformations also apply to fully-connected layers (as will be shown in §4).

To increase arithmetic intensity while performing the same number of FLOPs, one must decrease memory traffic, the denominator in Eqn. 3. Our key insight is that the total memory traffic of specialized CNNs with large batch size is dominated by reading/writing the input/output activations of convolutional and fully-connected layers ($NHWC_i$ and $NHWC_o$ in the denominator of Eqn. 3),[3] rather than by reading layer weights ($C_iK_HK_WC_o$). Figs. 4 and 5 (focus only on blue parts) depict this for one CNN: with batch size 1024, activations make up over 99% of total memory traffic.

Due to the dominance of input/output activations on a layer's total memory traffic, we note that a joint decrease in NHW and increase in $C_iK_HK_WC_o$ can reduce memory traffic while maintaining the same number of FLOPs. Suppose one decreased NHW by a factor of f (with $f > 1$) and increased C_i and C_o by a factor of

[3]The common practice of fusing activation functions to the preceding layer eliminates their contribution to total memory traffic.

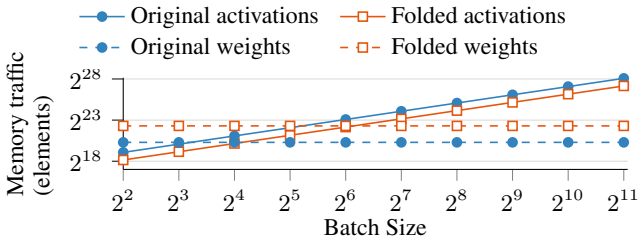


Figure 4. Memory traffic of activations and weights of the N1 and folded ($f = 4$) CNN. Axes are in log scale. The y-axis is in elements, rather than bytes, as the trends hold for any bitwidth.

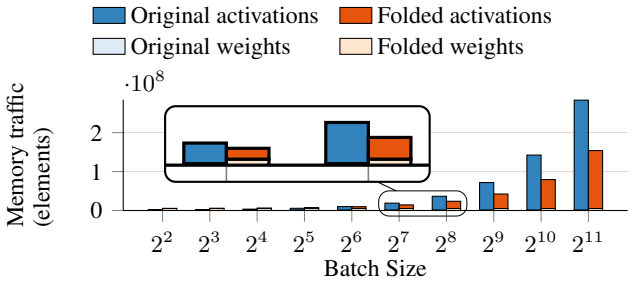


Figure 5. Total memory traffic of the N1 and folded ($f = 4$) CNN. As shown in the inset, weights account for a minor fraction of memory traffic with large batch size. The y-axis is in elements, rather than bytes, as the trends hold for any bitwidth used.

$\sqrt{f}$. We call this transformation *folding* and layers transformed by it *folded*. The folded layer has the following properties: (1) It performs the same number of FLOPs: $\frac{NHW}{f}(C_o\sqrt{f})(C_i\sqrt{f})(K_H K_W) = NHWC_oC_iK_HK_W$. (2) It decreases the size of layer inputs/outputs by a factor of $\sqrt{f}$ from $NHWC_i$ to $\frac{\sqrt{f}}{f}NHWC_i$ (similarly for outputs with C_o). (3) It increases the number of layer weights by a factor of f from $C_iK_HK_WC_o$ to $(C_i\sqrt{f})K_HK_W(C_o\sqrt{f})$.

Properties 2 and 3 are shown in Fig. 4 when folding a representative specialized CNN from the NoScope system with $f = 4$: the folded convolutions have 2× lower memory traffic for activations and 4× higher memory traffic for weights. At large batch sizes, the decrease in memory traffic for activations is larger than the increase for weights. For example, at batch size 1024, memory traffic for activations decreases by 66.7M, while that for weights increases by only 3.9M. The increase in memory traffic from layer weights is dwarfed by the decrease for activations, resulting in a reduction in total memory traffic. Fig. 5 illustrates this reduction in memory traffic for the same CNN. We analytically show when this reduction in memory traffic will occur in §J.

As the folded layer performs as many FLOPs as the original layer, but with reduced memory traffic, it has higher arithmetic intensity. If a layer is in the batch-limited regime, in which arithmetic intensity is determined by Eqn. 4, folding increases arithmetic intensity by $\sqrt{f}\times$, as the numerator and denominator in Eqn. 4 increase by $f\times$ and $\sqrt{f}\times$, respectively. An example of this is shown in §B.

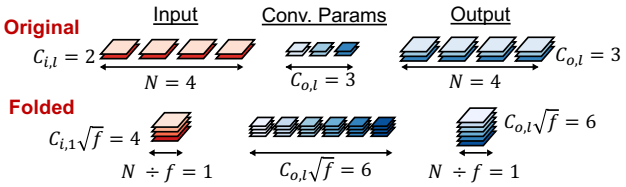


Figure 6. Middle layer of a FoldedCNN with $f = 4$. Both the number of input and output channels increase by a factor of $\sqrt{f}$.

When does folding help? Folding will most heavily increase the utilization and throughput of layers that have arithmetic intensity in the batch-limited regime that is below that needed for peak FLOPs/sec on an accelerator. Specialized CNNs are thus ideal targets for folding, as they have low arithmetic intensity even at large batch size. Meanwhile, large CNNs or those with small batch size are less likely to benefit. Thus, we focus on folding specialized CNNs.

4. FoldedCNNs

We now propose *FoldedCNNs*, a new approach CNN design based on the folding transformation proposed in §3.

Folding involves (1) decreasing NHW by $f\times$ and (2) increasing $C_iK_HK_WC_o$ by $f\times$. There are many ways to achieve these effects. FoldedCNNs achieve them by (1) decreasing batch size N by $f\times$, (2) increasing the number of input and output channels C_i and C_o each by $\sqrt{f}\times$. We do not reduce resolution (H, W) or increase receptive field (K_H, K_W), as specialized CNNs often operate over small images to begin with (Kang et al., 2017); we find that such changes can decrease accuracy compared to FoldedCNNs.

4.1. Applying folding to a full CNN

We now describe folding for a specialized CNN with L convolutional/fully-connected layers and C_L classes. Let $C_{i,l}$ denote the number of input channels to layer l of the original CNN, and $C'_{i,l}$ that in the FoldedCNN. Similar notation is used for all parameters in Table 2. While we focus on plain convolutions in this section, FoldedCNNs also apply to other convolutional variants. We evaluate folding for group convolutions in §C and Winograd convolutions in §D.

We first transform a layer l in the middle of the CNN, as shown in Fig. 6. As described above, FoldedCNNs decrease batch size: $N' = \frac{N}{f}$ and increase the number of input and output channels: $C'_{i,l} = C_{i,l}\sqrt{f}$ and $C'_{o,l} = C_{o,l}\sqrt{f}$. Folded fully-connected layers in the middle of the CNN also have $\sqrt{f}\times$ the number of input and output features. As folding is applied to all convolutional and fully-connected layers, the increase in output channels in one layer naturally fits the increase in input channels for the next layer.

Folding batches of images. As described in §3, each layer in a FoldedCNN performs the same number of FLOPs as the corresponding layer of the original CNN. However, a

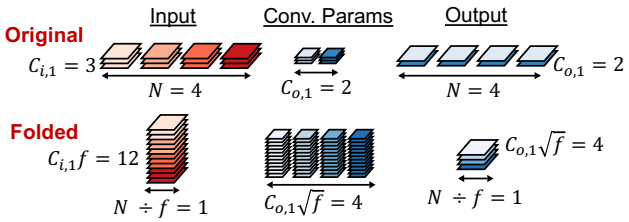


Figure 7. First layer of a FoldedCNN with $f = 4$. Unlike other layers, this layer increases input channels by a factor of f.

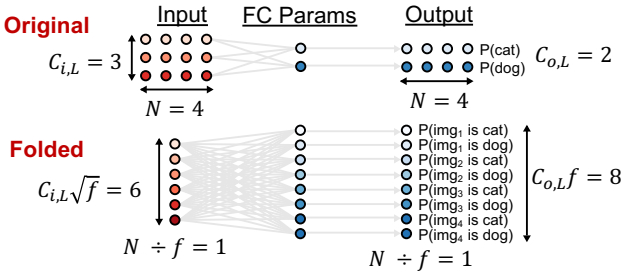


Figure 8. Output layer of a FoldedCNN with $f = 4$ and 2 classes. Unlike other layers, this layer has $f\times$ the number of outputs.

FoldedCNN performs these FLOPs over $\frac{N}{f}$ images, whereas the original CNN operates over N images. Left uncorrected, FoldedCNNs would thus perform $f\times$ more FLOPs per image, and thus would *reduce* application-level throughput.

To rectify this, FoldedCNNs "fold" a batch of images into "stacks" of images, as shown in Figs. 2 and 7. Suppose the original CNN takes in N images each with $C_{i,1}$ channels (e.g., $C_{i,1} = 3$ for RGB). A FoldedCNN instead takes in $\frac{N}{f}$ inputs each with $C_{i,1}f$ channels, formed by concatenating f images along the channels dimension. Each folded input represents f images, so the number of images in a batch of $\frac{N}{f}$ such inputs is equal to that of the original CNN (N). As a FoldedCNN performs inference over f images in a single input, it must return classification results for f images. To accommodate this, the output layer of a FoldedCNN produces outputs for fC_L classes, C_L for each of the f images stacked in a single input. This is illustrated in Fig. 8.

These adjustments result in the first and last layers of FoldedCNNs performing slightly more FLOPs than those of the original CNN. The first layer of a FoldedCNN sets $C'_{i,1} = C_{i,1}f$, whereas other layers have $C'_{i,l} = C_{i,l}\sqrt{f}$. As the number of output channels in the first layer is also increased by $\sqrt{f}\times$, the first layer performs $\sqrt{f}\times$ more FLOPs than the original first layer (see Fig. 7). This is also the case for the last layer of the FoldedCNN due to returning predictions for f images (see Fig. 8). All other layers in the FoldedCNN perform the same number of FLOPs as those in the original CNN, as described previously. Despite this slight increase in FLOPs, §5 will show that FoldedCNNs, in fact, *achieve higher throughput* than the original CNN due to their increased arithmetic intensity.

4.2. Training a FoldedCNN

Training a FoldedCNN is similar to training the original CNN. Let N_T denote the training batch size. Each training iteration, N_T images are sampled and transformed into $\frac{N_T}{f}$ folded inputs as described above. A forward pass through the FoldedCNN results in an output of size $\frac{N_T}{f} \times fC_L$, as shown in Fig. 8. This output is reshaped to be of size $N_T \times C_L$, and loss is computed on each of the N_T rows.

As each folded input consists of f images, and each image belongs to one of C_L classes, the effective number of classes for a FoldedCNN is C_L^f. This large increase in the number of classes can make it difficult to train a FoldedCNN for tasks with many classes to begin with. To combat this issue, we use a form of curriculum learning (Bengio et al., 2009) specialized for FoldedCNNs. Training begins by sampling from only $I < C_L$ classes of the original CNN's dataset, and introducing Δ more classes every E epochs. We hypothesize that starting with a small number of classes I avoids overloading the FoldedCNN with a difficult task early on in training, as $I^f \ll C_L^f$. We find this form of training beneficial when C_L and f are large, and it yielded only marginal improvements in other settings.

5. Evaluation

5.1. Evaluation setup

We consider CNNs and tasks from the usecases described in §2.1: specialized CNNs from NoScope[4] as lightweight filters, and specialized CNNs from Microsoft. Each task and CNN is described in detail in §A. While the focus of this work is on specialized CNNs, we also evaluate on the more general ResNet-18 on CIFAR-10 and CIFAR-100.

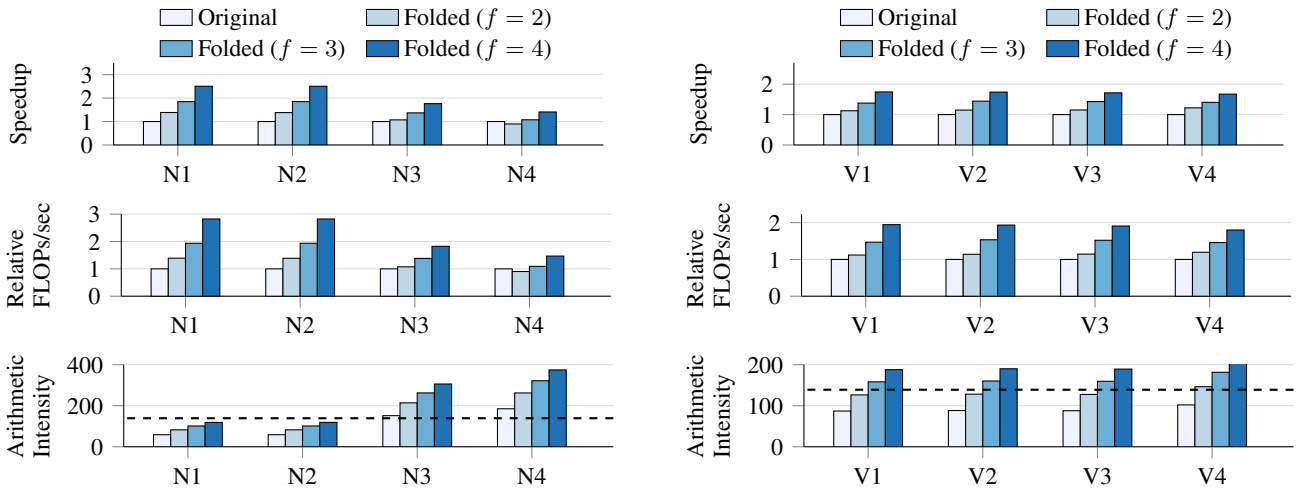
We evaluate FoldedCNNs with f of 2, 3, and 4, which increase the channels per layer by factors of roughly 1.41, 1.73, and 2, respectively ($\sqrt{f}\times$).[5] We compare FoldedCNNs to the compound scaling used in EfficientNets in §5.3.

Training setup. When training FoldedCNNs, we randomly assign images from the training set into groups of size f each epoch. Test sets are formed by randomly placing images from the test data into groups of f. Such randomization at test time avoids simpler settings, such folding f sequential frames in a video, thus providing a challenging scenario for FoldedCNNs. We also evaluate the sensitivity of FoldedCNNs to the order in which images are folded in §5.3.

We train all CNNs using cross entropy loss. Training takes place for for 50 epochs with batch size of 128 for the NoScope tasks and for 1500 epochs with batch size of 32 for

[4]Our evaluation focuses only on specialized CNNs, and thus does not reflect the performance of the full NoScope system.

[5]The number of channels resulting from folding are rounded down to avoid a non-integer number of channels (e.g., $\lfloor C_i\sqrt{f} \rfloor$).



(a) Specialized CNNs use in NoScope

(b) Specialized CNNs used at Microsoft

Figure 9. Inference performance of FoldedCNNs relative to the original CNN. Arithmetic intensity is plotted in absolute numbers, and the dashed line shows the minimum arithmetic intensity required to reach peak FLOPs/sec on a V100 GPU.

the game-scraping tasks. We use the curriculum learning in §4.2 for FoldedCNNs only on the game-scraping tasks. For these scenarios that use curriculum learning, we use $I = \max(f, \lfloor C_L/10 \rfloor)$, $\Delta = \lfloor C_L/10 \rfloor$, and $E = 60$. Such curriculum learning did not improve the accuracy of the original CNN. We use hyperparameters from NoScope (Kang et al., 2017) to train NoScope CNNs: RMSprop with learning rate 6.6×10^{-4} and Dropout of 0.25 after the second layer and before the last layer. All other models use Adam with learning rate 10^{-4} and weight decay of 10^{-5}.

Inference setup. We evaluate inference on a V100 GPU (p3.2xlarge AWS instance), which is typical of hardware used for specialized CNN inference in datacenters (Mullapudi et al., 2019). We also evaluate on T4 GPUs, which are common both in datacenters and edge clusters. As results on V100 and T4 are similar, we relegate results on T4 to §E. Inference is performed in PyTorch with TensorRT (NVIDIA, 2021) on CUDA 10.2. While FoldedCNNs can improve utilization for any numerical precision, we use half precision (FP-16) to use Tensor Cores, which offer the peak FLOPs/sec on the V100 (NVIDIA, 2017). We report utilization (FLOPs/sec) and application-level throughput (images/sec) relative to the original CNN via the mean of 10 trials of 10000 inferences of batch size 1024. We use other batch sizes in §F. We call relative throughput "speedup."

5.2. Evaluation on specialized CNNs used in NoScope

Utilization and throughput. Fig. 9a shows the speedup and FLOPs/sec of FoldedCNNs relative to the original CNN, and the arithmetic intensity of each CNN. FoldedCNNs increase FLOPs/sec by up to 2.8× and throughput by up to 2.5×. Increased throughput speeds up tasks like offline analytics, while increased utilization enables higher throughput on a single accelerator and a better return on investment for deploying accelerators. FoldedCNNs match the $\sqrt{f}\times$ theoretical increase in arithmetic intensity described in §3, thus increasing utilization and throughput with higher f.

FoldedCNNs result in larger improvements in utilization and throughput for the N1 and N2 CNNs (up to 2.8×) than for the N3 and N4 CNNs (up to 1.76×). This can be explained by arithmetic intensity: the N1 and N2 CNNs originally have very low arithmetic intensity. FoldedCNNs bring this arithmetic intensity much closer to that needed for peak performance on the V100 GPU, resulting in significantly higher utilization and throughput. In contrast, both N3 and N4 already have arithmetic intensity above the minimum needed for peak utilization, leaving less room for improvement. Despite this lower potential, FoldedCNNs still deliver up to 1.76× higher utilization and throughput for these CNNs.

There is only one case in which FoldedCNNs decrease throughput/utilization (N4, $f = 2$). This is due to tile quantization on NVIDIA GPUs, which we describe in §H.

Accuracy. Table 3 shows the accuracy of FoldedCNNs on the NoScope tasks. FoldedCNNs maintain high accuracy: the accuracy of FoldedCNNs with $f = 2$ is, in fact, higher than that of the original CNN for three of CNNs, and only 0.18% lower on the fourth. For these cases, FoldedCNNs provide up to a 1.39× speedup with the same accuracy.

As f increases, a FoldedCNN classifies more images per input, making the task of the FoldedCNN more challenging. As shown in Table 3 and Fig. 9a, increasing f reduces accuracy but increases utilization and throughput, introducing a tradeoff that can be spanned based on the requirements of applications. We analyze an example of this tradeoff in §G.

Table 3. Accuracy and speedup of FoldedCNNs for NoScope CNNs. Differences in accuracy are listed in parentheses.

	Original	FoldedCNN (f = 2)		FoldedCNN (f = 3)		FoldedCNN (f = 4)	
Model	**Accuracy**	**Accuracy**	**Speedup**	**Accuracy**	**Speedup**	**Accuracy**	**Speedup**
N1	98.82	98.64 (-0.18)	1.39	98.35 (-0.47)	1.85	97.93 (-0.89)	2.51
N2	96.96	96.99 (0.03)	1.38	96.93 (-0.03)	1.85	96.75 (-0.21)	2.50
N3	94.84	94.95 (0.11)	1.07	94.82 (-0.02)	1.37	94.72 (-0.12)	1.76
N4	91.66	91.91 (0.25)	0.90	91.39 (-0.27)	1.07	91.21 (-0.45)	1.41

Table 4. Performance of FoldedCNNs on production game-scraping tasks. Differences in accuracy are listed in parentheses.

			Original	FoldedCNN (f = 2)		FoldedCNN (f = 3)		FoldedCNN (f = 4)	
Model	**Resolution**	**Classes**	**Accuracy**	**Accuracy**	**Speedup**	**Accuracy**	**Speedup**	**Accuracy**	**Speedup**
V1	(22, 52)	11	97.64	97.64 (0.00)	1.13	97.18 (-0.46)	1.38	95.27 (-2.37)	1.75
V2	(19, 25)	22	93.45	92.09 (-1.36)	1.15	90.00 (-3.45)	1.44	89.91 (-3.54)	1.74
V3	(17, 40)	15	98.50	97.43 (-1.07)	1.15	97.20 (-1.30)	1.43	96.87 (-1.63)	1.71
V4	(22, 30)	27	96.52	96.52 (0.00)	1.22	96.00 (-0.52)	1.40	94.41 (-2.11)	1.67

Table 5. FoldedCNNs and EfficientNet compound scaling on game-scraping tasks. Speedup, utilization ("Util."), and arithmetic intensity ("A.I.") are relative to the original CNN.

		Higher values are better			
Model	**Mode**	**Accuracy**	**Speedup**	**Util.**	**A.I.**
V1	EfficientNet	93.27%	1.32	0.83	0.91
	Fold (f = 4)	95.27%	1.75	1.95	2.16
V2	EfficientNet	84.91%	1.51	0.80	0.88
	Fold (f = 4)	89.91%	1.74	1.93	2.15
V3	EfficientNet	96.40%	1.46	0.75	0.87
	Fold (f = 4)	96.87%	1.71	1.91	2.16
V4	EfficientNet	95.19%	1.34	0.83	0.91
	Fold (f = 3)	96.00%	1.40	1.46	1.78
	Fold (f = 4)	94.41%	1.67	1.80	2.10

5.3. Evaluation on production game-scraping CNNs

Fig. 9b shows the utilization, throughput, and arithmetic intensity of FoldedCNNs on the production game-scraping tasks. FoldedCNNs increase FLOPs/sec by up to 1.95× and throughput by up to 1.75× compared to the original CNN. Table 4 shows that FoldedCNNs have accuracy drops of 0–1.36%, 0.46–3.45%, and 1.63–3.54% with f of 2, 3, and 4 on these tasks. These drops are larger than those on the NoScope tasks due to the higher number of classes in the game-scraping tasks. While the NoScope tasks have only two classes, the game-scraping tasks have 11–27 classes. Thus, lower accuracy on the game-scraping tasks is expected from FoldedCNNs. That said, FoldedCNNs still enable large improvements, such as a 1.22× speedup with no accuracy loss for V4 with $f = 2$.

Effect of image order. As FoldedCNNs jointly classify f distinct images concatenated over the channels dimension, a natural question is how sensitive FoldedCNNs are to the order in which images are folded. To investigate this, we measure how often the predictions made by FoldedCNNs for each image match for all $f!$ permutations of f images folded together (e.g., how often do predictions for image X_1 match in folded inputs (X_1, X_2) and (X_2, X_1) for $f = 2$). With f of 2, 3, and 4, the average percentage of matching predictions for all $f!$ permutations on the V1 task is 98.8%, 98.4%, and 98.0%, showing high invariance to image order.

Comparison to EfficientNet scaling. We next compare FoldedCNNs to the techniques used in EfficientNets (Tan & Le, 2019). EfficientNets trade FLOPs and accuracy by jointly scaling the number of layers, the width, and the input resolution of a CNN. While such scaling can increase throughput by reducing FLOP count, reducing FLOP count in this manner can also decrease arithmetic intensity and utilization. To illustrate this, we transform the game-scraping CNNs with EfficientNet compound scaling[6] with the recommended parameters from the EfficientNet paper (Tan & Le, 2019): using terminology from the paper, $\phi = -1$, $\alpha = 1.1$, $\beta = 1.2$, and $\gamma = 1.15$. This transforms a CNN to perform roughly 2× fewer FLOPs, which increases throughput.

Table 5 compares FoldedCNNs and EfficientNets on the game-scraping CNNs. For each task, a FoldedCNN achieves both higher accuracy and throughput than the EfficientNet variant. For example, for V1, a FoldedCNN has 2% higher accuracy and 33% higher throughput than the EfficientNet variant. Furthermore, whereas EfficientNets reduce arithmetic intensity and utilization for all CNNs due to decreased FLOP count, FoldedCNNs uniformly increase arithmetic intensity and utilization. These results shows the promise of the new approaches proposed in FoldedCNNs targetted specifically for large-batch, specialized CNN inference.

5.4. FoldedCNNs in non-target settings

As described in §3, our focus in FoldedCNNs is on small CNNs with low arithmetic intensity even at large batch size, and specialized tasks with few classes. For completeness, we now evaluate FoldedCNNs on general-purpose CNNs and tasks, which are not in this target regime. We also evaluate small CNNs for tasks with many classes in §I.

[6] We do not use the EfficientNet-B0 architecture because it is significantly larger than typical specialized CNNs.

Accuracy on general tasks. To evaluate the accuracy of FoldedCNNs on general-purpose tasks, we consider ResNet-18 FoldedCNNs on CIFAR-10 and CIFAR-100.

For CIFAR-10, we train a FoldedCNN with $f = 4$ via distillation with the original CNN as the "teacher" (Ba & Caruana, 2014). The original ResNet-18 has an accuracy of 92.98%, while the FoldedCNN has an accuracy of 92.10%. This small accuracy drop even with high f shows the potential applicability of FoldedCNNs to general-purpose tasks.

For CIFAR-100, we do not observe benefit from the same distillation used for CIFAR-10. The original ResNet-18 on CIFAR-100 achieves 70.3% accuracy, while FoldedCNNs have accuracies of 68.11% (2.19% drop), 67.44% (2.86% drop), and 65.76% (4.54% drop) with f of 2, 3, and 4. These larger drops compared to CIFAR-10 can be attributed to the higher number of classes in CIFAR-100, which makes the task of a FoldedCNN more challenging (see §4.2).

Speedup on general CNNs. We now evaluate the speedup of FoldedCNNs when the original CNN is the general-purpose ResNet-18 operating on CIFAR-10. A FoldedCNN with $f = 4$ in this setup improves throughput by 8.1%. This speedup is smaller than those observed in Fig. 9 because ResNet-18 has arithmetic intensity of 430, much higher than the minimum needed for peak FLOPs/sec on a V100 (139). This places ResNet-18 outside the target regime of FoldedCNNs. FoldedCNNs still do provide 8.1% speedup, as 24% of the layers in ResNet-18 have low arithmetic intensity.

Takeaway. Coupling these moderate benefits in non-target settings with large benefits in target settings, FoldedCNNs show promise for increasing the utilization and throughput of specialized CNN inference beyond increased batch size.

6. Related Work

Efficient neural architectures. There is a large body of work on designing CNNs for efficient inference (e.g., (Ma et al., 2018; Cai et al., 2018; Zhou et al., 2018; Wu et al., 2019; Tan & Le, 2019; Cai et al., 2020)). Many of these works aim to reduce latency, but often do not consider accelerator utilization, which is a primary objective of FoldedCNNs. Some of these approaches, such as EfficientNets (Tan & Le, 2019), reduce the number of FLOPs performed by a CNN to achieve lower latency. However, we show in §5 that doing so can, in fact, reduce accelerator utilization. Furthermore, compared to these approaches, FoldedCNNs employ a fundamentally new structure to CNN inputs and classification, which could be integrated into existing architecture search techniques. Finally, FoldedCNNs are designed primarily for large-batch, specialized CNN inference, whereas existing works typically target general-purpose CNNs.

Improving throughput. Many other techniques have been proposed to accelerate inference, but which do not target utilization. Network pruning (Blalock et al., 2020) can improve throughput by reducing the FLOP count of a CNN, but, similar to the approaches described above, can reduce utilization. Reducing the numerical precision used during inference can increase throughput (Wang, 2019), but is insufficient for increasing utilization on modern accelerators (as we show in §2.3). Folding can be applied on top of these techniques to further improve the utilization and throughout of specialized CNN inference. In fact, our evaluation in §5 applies FoldedCNNs atop low-precision specialized CNNs.

Multitenancy. There is a growing body of work on increasing accelerator utilization by performing inference for multiple models on the same device (Narayanan et al., 2018; Jain et al., 2018; Shen et al., 2019; Yu & Chowdhury, 2020; Dhakal et al., 2020). These works do not improve the utilization of individual models, which is the goal of FoldedCNNs. Thus, these works are complementary to FoldedCNNs.

7. Conclusion

Specialized CNNs are widely used for high-throughput inference, but greatly underutilize accelerators, even when using large batch sizes. FoldedCNNs are a new approach to CNN design that increase the utilization and throughput of specialized CNN inference beyond increased batch size. FoldedCNNs increase arithmetic intensity by operating over distinct images concatenated along the channels dimension and increasing CNN width. Increased arithmetic intensity in FoldedCNNs boosts the utilization and throughput of specialized CNNs by up to 2.8× and 2.5×.

FoldedCNNs are not a panacea: their design is driven by the specific setting of specialized CNNs that operate over large batches, and that run on accelerators that require high arithmetic intensity to reach peak utilization. As our evaluation showed, FoldedCNNs provide only modest benefits outside this setting. Nevertheless, this work shows the increase in utilization and throughput made possible by substantially rethinking specialized CNN design. As the arithmetic intensity required to reach peak utilization on accelerators increases, FoldedCNNs may show promise in running today's general-purpose CNNs on tomorrow's accelerators.

Acknowledgements

We thank the anonymous reviewers and the area chair from ICML 2021, our colleagues at MSR, and CMU's Parallel Data Lab members for feedback that improved this work. We thank MSR for generously supporting Jack's internship work. Jack was also funded in part by an NSF Graduate Research Fellowship (DGE-1745016 and DGE-1252522). Jack and Rashmi were also funded in part by Amazon Web Services and in part by the AIDA project (POCI-01-0247-FEDER-045907) co-financed by the European Regional Development Fund through the Operational Program for Competitiveness and Internationalisation 2020.

References

AWS Outposts. https://aws.amazon.com/outposts/. Last accessed 08 June 2021.

Azure Stack Edge. https://azure.microsoft.com/en-us/products/azure-stack/edge/. Last accessed 08 June 2021.

Ba, J. and Caruana, R. Do Deep Nets Really Need to be Deep? In *Advances in Neural Information Processing Systems (NIPS 14)*, 2014.

Barroso, L. A., Clidaras, J., and Hölzle, U. The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, 2013.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. Curriculum Learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML 09)*, 2009.

Bhardwaj, R., Xia, Z., Ananthanarayanan, G., Jiang, J., Karianakis, N., Shu, Y., Hsieh, K., Bahl, V., and Stoica, I. Ekya: Continuous Learning of Video Analytics Models on Edge Compute Servers. *arXiv preprint arXiv:2012.10557*, 2020.

Blalock, D., Ortiz, J. J. G., Frankle, J., and Guttag, J. What is the State of Neural Network Pruning? In *The Third Conference on Systems and Machine Learning (MLSys 20)*, 2020.

Cai, H., Zhu, L., and Han, S. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *Proceedings of the 6th International Conference on Learning Representations (ICLR 18)*, 2018.

Cai, H., Gan, C., Wang, T., Zhang, Z., and Han, S. Once for All: Train One Network and Specialize it for Efficient Deployment. In *International Conference on Learning Representations (ICLR 20)*, 2020.

Canel, C., Kim, T., Zhou, G., Li, C., Lim, H., Andersen, D. G., Kaminsky, M., and Dulloor, S. R. Scaling Video Analytics on Constrained Edge Nodes. In *Proceedings of the 2nd Conference on Systems and Machine Learning (SysML 19)*, 2019.

Dhakal, A., Kulkarni, S. G., and Ramakrishnan, K. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SOCC 20)*, 2020.

Hsieh, K., Ananthanarayanan, G., Bodik, P., Venkataraman, S., Bahl, P., Philipose, M., Gibbons, P. B., and Mutlu, O. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

Jain, P., Mo, X., Jain, A., Subbaraj, H., Durrani, R. S., Tumanov, A., Gonzalez, J., and Stoica, I. Dynamic Space-Time Scheduling for GPU Inference. In *NeurIPS Workshop on Systems for Machine Learning*, 2018.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter Performance Snalysis of a Tensor Processing Unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA 17)*, 2017.

Kang, D., Emmons, J., Abuzaid, F., Bailis, P., and Zaharia, M. NoScope: Optimizing Neural Network Queries over Video at Scale. *Proceedings of the VLDB Endowment*, 10 (11):1586–1597, 2017.

Kang, D., Bailis, P., and Zaharia, M. BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics. *Proceedings of the VLDB Endowment*, 13(4), 2020.

Kosaian, J., Rashmi, K. V., and Venkataraman, S. Learning-Based Coded Computation. *IEEE Journal on Selected Areas in Information Theory*, 2020.

Ma, N., Zhang, X., Zheng, H.-T., and Sun, J. ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. In *Proceedings of the 15th European Conference on Computer Vision (ECCV 18)*, 2018.

Mullapudi, R. T., Chen, S., Zhang, K., Ramanan, D., and Fatahalian, K. Online Model Distillation for Efficient Video Inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV 19)*, pp. 3573–3582, 2019.

Narayanan, D., Santhanam, K., Phanishayee, A., and Zaharia, M. Accelerating Deep Learning Workloads Through Efficient Multi-Model Execution. In *NeurIPS Workshop on Systems for Machine Learning*, 2018.

NVIDIA. NVIDIA Deep Learning Performance Guide. https://docs.nvidia.com/deeplearning/sdk/dl-performance-guide/index.html. Last accessed 08 June 2021.

NVIDIA. NVIDIA Tesla V100 GPU Architecture. Technical Report WP-08608-001_v1.1, 2017.

NVIDIA. NVIDIA Turing GPU Architecture. Technical Report WP-09183-001_v01, 2018.

NVIDIA. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt, 2021. Last accessed 08 June 2021.

Shen, H., Han, S., Philipose, M., and Krishnamurthy, A. Fast Video Classification via Adaptive Cascading of Deep Models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 17)*, 2017.

Shen, H., Chen, L., Jin, Y., Zhao, L., Kong, B., Philipose, M., Krishnamurthy, A., and Sundaram, R. Nexus: A GPU Cluster Engine for Accelerating DNN-based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 19)*, 2019.

Tan, M. and Le, Q. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML 19)*, 2019.

Wang, H. Low Precision Inference on GPU. https://tinyurl.com/1g9e5dpw, 2019. Last accessed 08 June 2021.

Williams, S., Waterman, A., and Patterson, D. Roofline: an Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., Tian, Y., Vajda, P., Jia, Y., and Keutzer, K. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 19)*, 2019.

Yu, P. and Chowdhury, M. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *The Third Conference on Systems and Machine Learning (MLSys 20)*, 2020.

Zhou, Y., Ebrahimi, S., Arık, S. Ö., Yu, H., Liu, H., and Diamos, G. Resource-Efficient Neural Architect. *arXiv preprint arXiv:1806.07912*, 2018.

A. Datasets

A.1. Datasets and models used in game-scraping application

This section provides details on the datasets and models used in the production video-game-scraping workload described in §2. The images in each dataset represent the style of text that will appear in a particular portion of a game screen, which will be used in a downstream event detection pipeline.

Dataset generation. The location and style of relevant text in a particular video game may differ from stream-to-stream. To avoid the need to manually label streams, the game-scraping application generates synthetic datasets for training, validation, and testing.

Specifically, the text that will appear in images for a particular dataset follows a predefined structure. For example, the text appearing in images of the V1 task is of the form "XY.Zk", where X, Y, and Z each represent a digit 0 through 9, and k is the string literal "k". From these specifications, examples that match certain classes of a particular dataset can be generated. For example, V1 classifies the Z digit in the specification above, and might generate "67.8k" and "04.8k" as instances of this specification for class "8".

Once an instance of a specification has been constructed, an image containing this text is generated. In order to train a model that is robust to perturbations in text location, text font, and background color/texture, the generation process selects fonts, locations, and backgrounds for the generated image at random from a set of prespecified options. Figures 10–15 below show the effects of this randomization.

We now provide details of each dataset used for this task in the paper. Example images chosen randomly from the validation sets of each dataset are displayed. We also describe the detailed architecture of the specialized CNNs employed for each dataset. For brevity, we use the following notation to describe CNNs: CX is a 3×3 2D convolution with X output channels and stride of 1, M is a 2D max pool with kernel size 3×3 and stride of 1, FX is a fully-connected layer with X output features. We use B as shorthand notation for C32 $\rightarrow$ C32 $\rightarrow$ C32. ReLUs follow each convolutional layer and all but the final fully-connected layer.

Figure 10. Example images in the lol-gold1 dataset.

V1: lol-gold1.

- **Game:** League of Legends
- **Number of classes:** 11
- **Image resolution:** (22, 52)
- **Example:** Figure 10
- **Model:** C32 $\rightarrow$ M $\rightarrow$ B $\rightarrow$ M $\rightarrow$ C8 $\rightarrow$ M $\rightarrow$ F11
- **Description:** Classifies the fractional value of a count of the amount of gold a player has accumulated (e.g., "7" in "14.7k"). Classes are digits 0 through 9 and "other" indicating that the section is blank.
- **Training images per class:** 10000
- **Validation images per class:** 100
- **Test images per class:** 100

Figure 11. Example images in the apex-count dataset.

V2: apex-count.

- **Game:** Apex Legends
- **Number of classes:** 22
- **Image resolution:** (19, 25)
- **Example:** Figure 11
- **Model:** C32 $\rightarrow$ M $\rightarrow$ B $\rightarrow$ M $\rightarrow$ C8 $\rightarrow$ M $\rightarrow$ F22
- **Description:** Classifies the number of members of a squad remaining. Classes are integers 0 through 20 and "other" indicating that the section is blank.
- **Training images per class:** 1000
- **Validation images per class:** 100
- **Test images per class:** 100

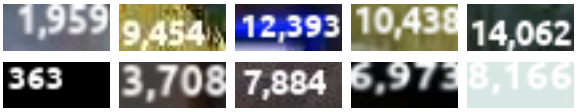


Figure 12. Example images in the sot-coin dataset.

V3: sot-coin.

- **Game:** Sea of Thieves
- **Number of classes:** 15
- **Image resolution:** (17, 40)
- **Example:** Figure 12
- **Model:** C32 $\rightarrow$ M $\rightarrow$ B $\rightarrow$ M $\rightarrow$ C8 $\rightarrow$ M $\rightarrow$ F15
- **Description:** Classifies the thousands-place of a count on the number of coins a player has (e.g., "10" for "10,438"). Classes are integers 0 through 14 and "other" indicating that the section is blank.
- **Training images per class:** 800

- **Validation images per class:** 200
- **Test images per class:** 200

Figure 13. Example images in the sot-time dataset.

V4: sot-time.

- **Game:** Sea of Thieves
- **Number of classes:** 27
- **Image resolution:** (22, 30)
- **Example:** Figure 13
- **Model:** C32 → M → B → M → B → M → C8 → M → F27
- **Description:** Classifies the time remaining. Classes are integers 0 through 25 and "other" indicating that the section is blank.
- **Training images per class:** 2000
- **Validation images per class:** 100
- **Test images per class:** 100

Figure 14. Example images in the lol-gold2 dataset.

V5: lol-gold2.

- **Game:** League of Legends
- **Number of classes:** 111
- **Image resolution:** (22, 52)
- **Example:** Figure 14
- **Model:** C32 → M → B → M → C8 → M → F111
- **Description:** Classifies the integer value of a count of the amount of gold a player has accumulated (e.g., "14" in "14.7k"). Classes are digits 0 through 9, 00 through 99, and "other" indicating that the section is blank.
- **Training images per class:** 1000
- **Validation images per class:** 100
- **Test images per class:** 100
- **Note:** This CNN is used only in §I

V6: lol-time.

- **Game:** League of Legends
- **Number of classes:** 62
- **Image resolution:** (15, 35)

Figure 15. Example images in the lol-time dataset.

- **Example:** Figure 15
- **Model:** C32 → M → B → M → C8 → M → F62
- **Description:** Classifies the minutes place of a timer (e.g., "30" in "30:54"). Classes are digits 00 through 60 and "other" indicating that the section is blank.
- **Training images per class:** 1000
- **Validation images per class:** 100
- **Test images per class:** 100
- **Note:** This CNN is used only in §I

A.2. Datasets and models used in NoScope

We evaluate folding using four of the datasets from NoScope (Kang et al., 2017). Each task involves binary classification of whether an object of interest is present in a frame. As the overall videos provided contain millions of frames, we sample contiguous subsets of frames in the video to form training, validation, and testing sets.

The CNNs used in evaluation follow those described in the NoScope paper and source code. We next detail these architectures, as well as the splits of the dataset used in evaluation.

N1: coral.

- **Object of interest:** person
- **Model:** C16 → C16 → M → F128 → F2
- **Total video duration:** 11 hrs.
- **Dataset split:** Split the video into eight contiguous chunks. Use chunk 6 as a training dataset, chunk 7 as a validation dataset, and chunk 8 as a testing dataset.

N2: night.

- **Object of interest:** car
- **Model:** C16 → C16 → M → F128 → F2
- **Total video duration:** 8.5 hrs.
- **Dataset split:** Split the video into eight contiguous chunks. Use chunk 2 as a training dataset, chunk 3 as a validation dataset, and chunk 4 as a testing dataset.

N3: roundabout.

- **Object of interest:** car
- **Model:** C32 → C32 → M → C64 → C64 → M → F32 → F2
- **Total video duration:** 8.1 hrs.

- **Dataset split:** Split the video into eight contiguous chunks. Use chunk 2 as a training dataset, chunk 3 as a validation dataset, and chunk 4 as a testing dataset.

N4: taipei.

- **Object of interest:** bus
- **Model:** C64 → C64 → M → F32 → F2
- **Total video duration:** 12 hrs.
- **Dataset split:** Split the video into sixteen contiguous chunks. Use the chunk 1 as a training dataset, chunk 2 as a validation dataset, and chunk 3 as a testing dataset.

B. Example of Folding a Single Layer

Table 6 shows an example of folding a convolutional layer from a specialized CNN used in the game-scraping workload. The memory traffic of the original layer is dominated by the input and output activations of the layer. Folding with $f = 4$ reduces memory traffic by nearly $2\times$ while maintaining the same number of operations, enabling a $2\times$ increase in arithmetic intensity.

C. Folding for Group Convolutions

In this section, we describe how folding is applied to group convolutions.

Background on group convolutions. In a group convolution, the input and output channels of the convolution are split into G groups. Each output channel in a particular group is computed via convolution over only those input channels in the corresponding group. This results in a G-fold decrease in operations and a G-fold decrease in the number of parameters in the convolutional layer. The resultant arithmetic intensity for a group convolution is thus:

$$\frac{2NHWC_oC_iK_HK_W/G}{B(NHWC_i + \frac{C_iK_HK_WC_o}{G} + NHWC_o)}$$

The arithmetic intensity of a group convolution in the batch-limited regime (defined in §2.3) is determined as follows (recalling from §2.3 that calculating arithmetic intensity in the batch-limited regime involves removing the variable B):

$$A = \frac{2NHWC_oC_iK_HK_W/G}{NHWC_i + \frac{C_iK_HK_WC_o}{G} + NHWC_o}$$

$$\lim_{N\to\infty} A = \frac{2C_oC_iK_HK_W}{C_i + C_o} * \frac{1}{G}$$

Comparing this arithmetic intensity to that in Eqn. 4 of the main paper, the arithmetic intensity of a group convolution with G groups in the batch-limited regime is $G\times$ lower than a corresponding "vanilla" convolution. This makes group convolutions a promising target for increasing arithmetic intensity via folding.

Applying folding to group convolutions. Folding group convolutions is straightforward. Similar to folding "vanilla" convolutions, a FoldedCNN for a group convolution with C_i input channels and C_o output channels reduces batch size by a factor of f and increases C_i and C_o each by a factor of $\sqrt{f}\times$. This results in increasing the number of channels per group in the group convolution by a factor of $\sqrt{f}$, and thus also increases arithmetic intensity in the batch-limited regime by a factor of $\sqrt{f}$.

Inference performance of folded group convolutions. We evaluate the throughput and utilization of folding on two group convolutions shown in Table 7. The two group convolutions are identical other than the number of total input and output channels, with G32 having 32 and G64 having 64. Each setting uses 4 groups, leading to 8 and 16 channels per group for G32 and G64, respectively. We compare the throughput and utilization of these convolutions to the corresponding folded version with $f = 4$. Folding results in 64 input and output channels with 16 channels per group for G32, and 128 input and output channels with 32 channels per group for G64. We use the same experimental setup described in §5 of the main paper for evaluating inference performance.

With batch size of 1024, folding with $f = 4$ increase throughput and utilization of these grouped convolutions by $1.74\times$ for G32 and by $1.59\times$ for G64 on a V100 GPU. Folding increases arithmetic intensity by nearly a factor of two for each convolution. The larger improvement for G32 compared to G64 comes from the lower arithmetic intensity of G32; due to having half the number of input and output channels of G64, G32 has half the arithmetic intensity. Thus, there is more room for improving the utilization of G32 by increasing arithmetic intensity alone via folding. These results show the effectiveness of folding on group convolutions.

D. Folding for Winograd Convolutions

FoldedCNNs can benefit a wide variety of convolutional implementations, such as direct convolutions, matrix-multiplication-based convolutions, and Winograd convolutions. In fact, our evaluation in §5 runs atop TensorRT, which selects among convolutional implementations, including Winograd. To more clearly illustrate the performance of FoldedCNNs on Winograd convolutions, we also directly run FoldedCNNs using Winograd convolutions in cuDNN. Here, on the video scraping CNNs using the same experimental setup described in §5, FoldedCNNs with $f = 4$ provided a median speedup of $1.66\times$ over the original CNN, matching the speedups in §5.3.

Table 6. Example of increasing arithmetic intensity by folding a convolutional layer with $f = 4$. The layer has $K_H = K_W = 3$, $H = 11$, $W = 26$, and uses half precision (i.e., $B = 2$).

	Original		Folded (f = 4)	
	Equation	**Value**	**Equation**	**Value**
Batch size	N	1024	N/f	256
Input, output channels	C_i, C_o	32, 32	$C_i\sqrt{f}, C_o\sqrt{f}$	64, 64
Input and output elements (E_{io})	$NHWC_i + NHWC_o$	18.74M	$\frac{\sqrt{f}}{f}NHWC_i + \frac{\sqrt{f}}{f}NHWC_o$	9.37M
Layer elements (E_l)	$C_iK_HK_WC_o$	0.01M	$fC_iK_HK_WC_o$	0.04M
Memory traffic in bytes (M)	$B(E_{io} + E_l)$	37.51M	$B(E_{io} + E_l)$	18.82M
Operations (O)	$2NHWC_oC_iK_HK_W$	5398.07M	$2NHWC_oC_iK_HK_W$	5398.07M
Arithmetic intensity	O/M	**143.93**	O/M	**286.87**

Table 7. Group convolutions evaluated

Name	C_i	C_o	G	K_H	K_W	H	W
G32	32	32	4	3	3	50	50
G64	64	64					

E. Inference Performance on T4 GPU

Figure 16 shows the throughput, FLOPs/sec, and arithmetic intensity achieved by FoldedCNNs and the original CNN on a T4 GPU (AWS g4dn.xlarge instance) in half-precision. The general trends are similar to those described in §5 of the main paper for the V100 GPU.

F. Speedup with Varying Batch Size

Figure 17 shows the throughput improvement when using FoldedCNNs with various values of f relative to the original CNN at varying batch sizes. As shown in the figure, the throughput improvement resulting from folding is largest at a batch size of 2048, and decreases with decreasing batch size. This behavior is expected, as decreasing batch size N decreases the likelihood that the inequality proved in §J will hold, and thus that folding will benefit. Folding is designed for improving high-throughput specialized CNN inference, in which large batch sizes are used.

G. Accuracy-Throughput Tradeoff

When reasoning about the potential tradeoff between accuracy and throughput/utilization present with FoldedCNNs, it is important to consider the usecases of specialized CNNs. As described in §2.1, it is common to use specialized CNNs as a lightweight filter in front of a large, general-purpose CNN. In such systems, most inputs are processed only by the specialized CNN, rather than by both the specialized CNN and the general-purpose CNN. Thus, the throughput of the specialized CNN typically dominates the total throughput of the system.

Given the heavy use of the specialized CNN in this setup, improving the throughput of the specialized CNN at the expense passing more inputs to the general-purpose CNN may increase system throughput. For example, a FoldedCNN with $f = 4$ speeds up the N2 CNN by 2.50× with a 0.21% drop in accuracy. We show below that this FoldedCNN increases system throughput unless the general-purpose CNN is over 285× slower than the N2 CNN. Thus, the improved utilization and throughput of specialized CNNs made possible by FoldedCNNs can compensate for reduced their accuracy to improve total system throughput.

We now walk through this accuracy-throughput tradeoff via an abstract example. Figure 18 shows an abstract example of using a specialized CNN (e.g., those from NoScope) as a lightweight filter in front of a large, general-purpose CNN (e.g., ResNet-50). As depicted in the figure, all inputs pass through the specialized CNN, which has a latency of T_s. The specialized CNN is unsure about u fraction of those inputs, and thus forwards the inputs to the general-purpose CNN, which has a latency of T_g. For the remaining $(1 - u)$ fraction of inputs, the specialized CNN is sure of its answer, and returns the prediction directly.

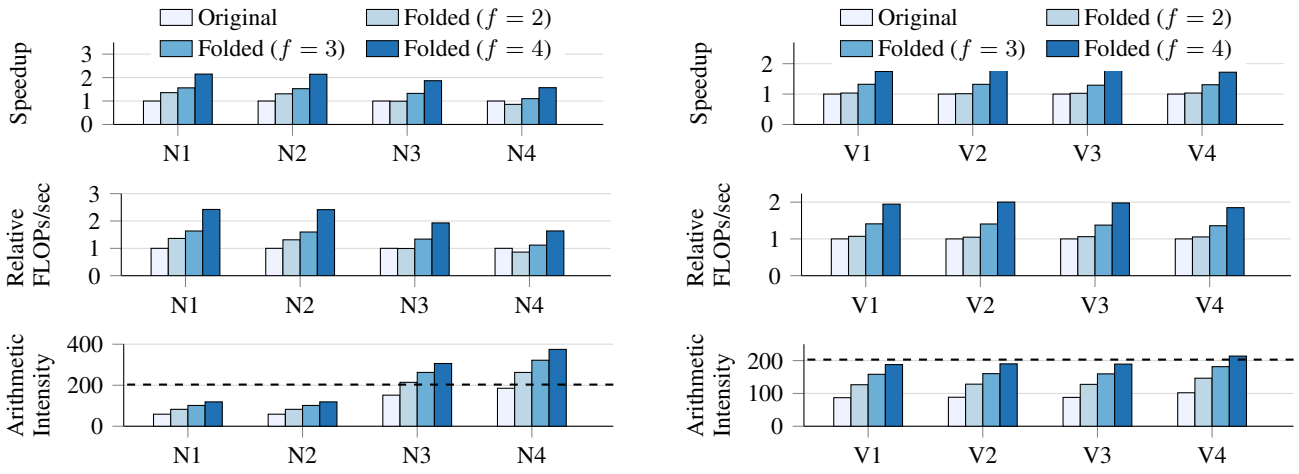
The expected latency for a given input to this system is thus:

$$E[T] = T_s + uT_g$$

Suppose that one replaced the specialized CNN used in such an application with a FoldedCNN that increases throughput by a factor of x, but decreases accuracy by a. Under the reasonable assumption that an increase in throughput leads to a corresponding decrease in latency, the latency of the FoldedCNN can be given as $\frac{T_s}{x}$. Furthermore, under the assumption that all incorrectly classified inputs from the specialized CNN are forwarded to the general-purpose CNN (i.e., u is equivalent to the error of the specialized CNN), then the FoldedCNN lets $u + a$ fraction of frames through to the general-purpose CNN. Thus, the expected latency for a given input to the system with a FoldedCNN is:

$$E[T] = \frac{T_s}{x} + (u + a)T_g$$

Clearly, for high values of x and small values a, the FoldedCNN can result in improved total system throughput (recip-



(a) NoScope specialized CNNs

(b) Game-scraping specialized CNNs at a cloud-service provider

Figure 16. Inference performance of FoldedCNNs relative to the original CNN. Arithmetic intensity is plotted in absolute numbers, and the dashed line shows the minimum arithmetic intensity required to reach peak FLOPs/sec on a T4 GPU.

rocal of latency). A secondary question of interest is: given specific values of x and a, for what values of T_s and T_g does the FoldedCNN increase overall system throughput?

To answer this question, we focus on the ratio $\frac{T_g}{T_s}$. Intuitively, the higher this ratio, the larger the effect of inaccuracy of the FoldedCNN on overall system throughput. We next calculate the maximum value this ratio can be for a FoldedCNN to improve overall system throughput:

$$
\begin{aligned}
T_s + uT_g &> \frac{T_s}{x} + (u+a)T_g \\
T_s - \frac{T_s}{x} &> (u+a)T_g - uT_g \\
T_s(1 - \frac{1}{x}) &> aT_g \\
\frac{1}{a}(1 - \frac{1}{x}) &> \frac{T_g}{T_s}
\end{aligned}
$$

Consider the FoldedCNN with $f = 4$ for the N2 dataset described in §5.2 of the main paper. This FoldedCNN results in an increase in throughput of $x = 2.5\times$ and a decrease in accuracy of $a = 0.0021$. Plugging these values into the inequality above shows that this FoldedCNN will result in an overall improvement in system throughput so long as the general-purpose CNN is less than $285\times$ slower than the original specialized CNN. If we consider ResNet-50 as an example of a general-purpose CNN, this is easily satisfied for the N2 CNN: ResNet-50 is $83\times$ slower than the original specialized CNN.

H. Effect of Tile Quantization on the Performance of FoldedCNNs

In §5.2, we observed one case in which a FoldedCNN resulted in a decrease in throughput and utilization compared to the original CNN: the N4 CNN using $f = 4$. After investigating the CNN, we found the cause to be due to *GPU tile quantization*: when the problem size does not divide evenly into a chosen tile size (i.e., the size of partitions of the overall kernel) (NVIDIA). NVIDIA's deep learning libraries are best optimized for cases in which certain parameters of a convolutional layer, such as input and output channels, are divisible by large powers of two (e.g., divisible by 64 or 128) (NVIDIA). Parameters that do not meet this requirement typically use kernels optimized for the next highest number divisible by a large power of two, resulting in a significant amount of wasted work. For more details on the inefficiency resulting from tile quantization, please see NVIDIA's deep learning performance guide (NVIDIA).

Many CNNs are already designed to have a number of input and output channels that are a power of two (e.g., many of the specialized CNNs have convolutional layers with 32 input and output channels). However, FoldedCNN's increase the number of input and output channels by a factor of $\sqrt{f}$. For non-square values of f, such as 2 and 3, applying folding to such a layer may result in a number of input or output channels that is no longer a power of two or is no longer divisible by a power of two. For example, applying folding with $f = 2$ to a convolutional layer with 64 input and output channels will result in a convolutional layer with $\lfloor 64\sqrt{2} \rfloor = 90$ channels.

For the values of f considered in this work, we find that tile quantization primarily affects convolutions with a number

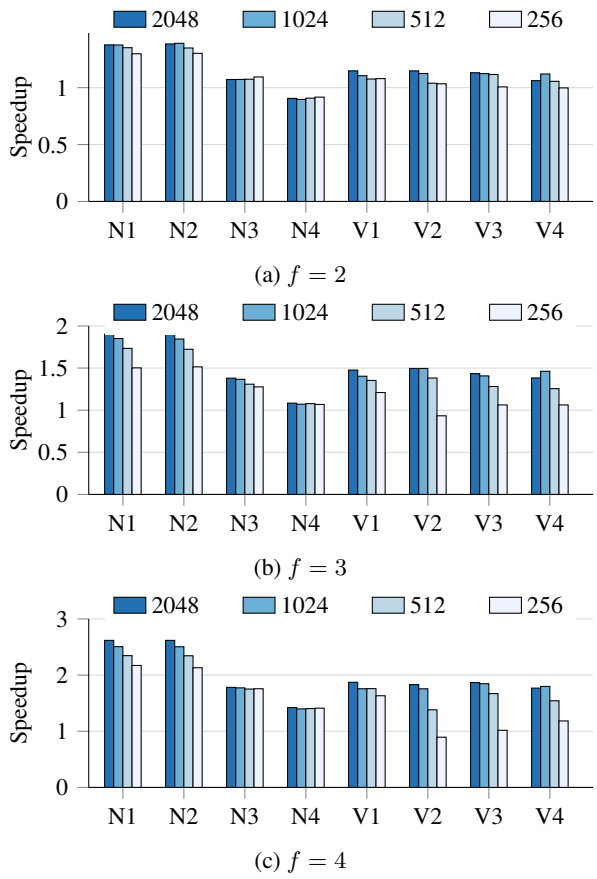


Figure 17. Speedup of FoldedCNNs with varying f at various batch sizes relative to the throughput of the original CNN at corresponding batch sizes.

of input and output channels greater than or equal to 64; we do not observe the negative effects often associated with tile quantization for convolutions with fewer channels, such as 32 or 16.

As shown in §A, the N4 CNN contains two convolutions with 64 intermediate channels, followed by a fully-connected layer with 32 output neurons. The FoldedCNN with f of 2 and 3 will thus lead to the negative effects of tile quantization for the convolutions in this CNN, but not for the fully-connected layer, which will receive the full benefits of folding. With $f = 2$, the benefit from folding does not outweigh the inefficiency due to tile quantization, resulting in a net decrease in utilization and throughput. In contrast, with $f = 3$, the benefits of folding outweigh the cost of tile quantization, resulting in an increase in utilization and throughput, albeit less pronounced than expected for $f = 3$.

It is important to note that this case with decreased utilization and throughput is *not due to incorrectness of the transformations performed by FoldedCNNs.* As shown in §5.2 of the main paper, FoldedCNNs with f of 2 and 3 for the N4 CNN result in the expected $\sqrt{f}\times$ improvements in arithmetic intensity. Decreased inference performance in this case is due to lower levels of the system software (e.g., TensorRT, cuDNN), rather than the design of FoldedCNNs themselves. Other accelerators may not face the same issue.

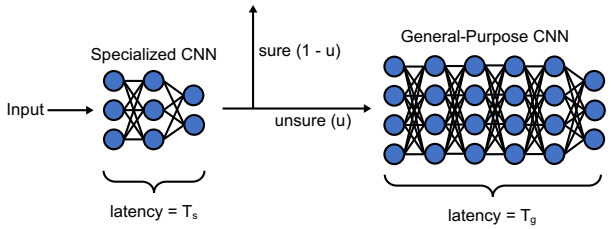


Figure 18. Abstract example of the use of a specialized CNN as a lightweight filter in front of a larger, general-purpose CNN.

I. Evaluation on Small CNNs with many Classes

In this section, we consider CNNs that have the same size as specialized CNNs, but which operate over many classes. We consider two new game-scraping tasks: a task with 111 classes (V5), and one with 62 classes (V6[7]). We use the same CNN as that used for V1. Table 8 shows that FoldedCNNs exhibit larger drops in accuracy on these tasks due to the larger number of classes, but still increase utilization/throughput by up to 1.75×.

J. Proof of Reduction in Memory Traffic

We will prove that the transformation described in §3 reduces total memory traffic if:

$$NHW > \frac{(f-1)C_i K_H K_W C_o}{(1-\frac{1}{\sqrt{f}})(C_i + C_o)} \tag{5}$$

Recall that the arithmetic intensity of a convolutional layer as given by Eqn. 3 is:

$$\frac{\text{FLOPs}}{\text{Bytes}} = \frac{2NHWC_oC_iK_HK_W}{B(NHWC_i + C_iK_HK_WC_o + NHWC_o)}$$

In §3, we create a new version of the layer in which we decrease NHW by a factor of f and increase both C_i and C_o by a factor of $\sqrt{f}$. We wish to show that this new layer has a higher arithmetic intensity than the original layer,

[7]For this CNN, we find that the small input resolution and large number of classes requires using more specially-tuned curriculum learning parameters. Specifically, when training a FoldedCNN with $f = 4$ on this dataset, we use $I = 4$, $\Delta = 3$, and $E = 120$, and train the CNN for 3000 epochs.

Table 8. Performance of FoldedCNNs for CNNs with many classes. Differences in accuracy are listed in parentheses.

			Original	Folded (f = 2)		Folded (f = 3)		Folded (f = 4)	
Model	**Resolution**	**Classes**	**Accuracy**	**Accuracy**	**Speedup**	**Accuracy**	**Speedup**	**Accuracy**	**Speedup**
V5	(22, 52)	111	93.95	92.50 (-1.45)	1.12	90.78 (-3.17)	1.42	87.51 (-6.44)	1.75
V6	(15, 35)	62	89.71	88.03 (-1.68)	1.08	85.48 (-4.23)	1.42	84.92 (-4.79)	1.71

provided that:

$$NHW > \frac{(f-1)C_iK_HK_WC_o}{(1-\frac{1}{\sqrt{f}})(C_i+C_o)}$$

We will first show that the new layer performs an equal number of operations as the original layer (i.e., the numerator in Eqn. 3 stays the same) and then show that the new layer layer has reduced memory traffic compared to the original layer (i.e., the denominator in Eqn. 3 decreases), provided that the inequality holds. These two changes will result in the new layer having an increased arithmetic intensity.

Equal number of operations. The initial convolutional layer performs $2NHWC_oC_iK_HK_W$ operations. The transformed convolutional layer performs $\frac{2NHW}{f}(\sqrt{f}C_o)(\sqrt{f}C_i)(K_HK_W) = 2NHWC_oC_iK_HK_W$ operations, which is equal to that of the original model.

Reduced memory traffic. We wish to show that the inequality is equivalent to the memory traffic of the transformed layer being lower than that of the original layer.

We first note that, ignoring the bytes per element B, the memory traffic of the original convolutional layer is $NHWC_i + C_iK_HK_WC_o + NHWC_o$, while that of the transformed layer is $\frac{\sqrt{f}}{f}NHWC_i + fC_iK_HK_WC_o + \frac{\sqrt{f}}{f}NHWC_o$.

We wish to show that:

$$NHWC_i + C_iK_HK_WC_o + NHWC_o > \frac{\sqrt{f}}{f}NHWC_i + fC_iK_HK_WC_o + \frac{\sqrt{f}}{f}NHWC_o$$

We first rearrange the righthand side of the inequality as:

$$\frac{\sqrt{f}}{f}NHWC_i + fC_iK_HK_WC_o + \frac{\sqrt{f}}{f}NHWC_o = \frac{NHW}{\sqrt{f}}C_i + fC_iK_HK_WC_o + \frac{NHW}{\sqrt{f}}C_o$$

Grouping by similar terms gives:

$$NHWC_i - \frac{NHW}{\sqrt{f}}C_i + NHWC_o - \frac{NHW}{\sqrt{f}}C_o > fC_iK_HK_WC_o - C_iK_HK_WC_o$$

Which implies:

$$(1-\frac{1}{\sqrt{f}})NHWC_i + (1-\frac{1}{\sqrt{f}})NHWC_o > (f-1)C_iK_HK_WC_o$$

Which implies:

$$NHW((1-\frac{1}{\sqrt{f}})(C_i+C_o)) > (f-1)C_iK_HK_WC_o$$

Which ultimately leads to our desired inequality:

$$NHW > \frac{(f-1)C_iK_HK_WC_o}{(1-\frac{1}{\sqrt{f}})(C_i+C_o)}$$