EVA^2: Exploiting Temporal Redundancy in Live Computer Vision

Mark Buckler
Cornell University

Philip Bedoukian
Cornell University

Suren Jayasuriya
Arizona State University

Adrian Sampson
Cornell University

ABSTRACT

Hardware support for deep convolutional neural networks (CNNs) is critical to advanced computer vision in mobile and embedded devices. Current designs, however, accelerate generic CNNs; they do not exploit the unique characteristics of real-time vision. We propose to use the temporal redundancy in natural video to avoid unnecessary computation on most frames. A new algorithm, *activation motion compensation*, detects changes in the visual input and incrementally updates a previously-computed output. The technique takes inspiration from video compression and applies well-known motion estimation techniques to adapt to visual changes. We use an adaptive key frame rate to control the trade-off between efficiency and vision quality as the input changes. We implement the technique in hardware as an extension to existing state-of-the-art CNN accelerator designs. The new unit reduces the average energy per frame by 54.2%, 61.7%, and 87.6% for three CNNs with less than 1% loss in vision accuracy.

1. INTRODUCTION

Convolutional neural networks (CNNs) have revolutionized computer vision. A commensurate flurry of architectural work has designed efficient hardware CNN implementations [1, 2, 3, 4] targeting resource-strapped mobile and embedded systems. These designs, however, target generic convolutional networks: they do not exploit any domain-specific characteristics of embedded vision to improve CNN execution. This paper specializes CNN hardware for real-time vision on live video, enabling efficiency benefits unavailable to generic accelerators.

The key insight is that live vision is *temporally redundant*. In an input video, every frame differs only slightly from previous frames. A generic CNN accelerator, however, runs nearly identical, equally expensive computations for every frame. This traditional strategy wastes work to compute similar outputs for visually similar inputs. Existing work has also shown that CNNs are *naturally approximate*. Sparse accelerators, for example, improve efficiency with negligible output impact by rounding small network weights down to zero [3, 4, 5, 6].

To exploit both temporal redundancy and approximation in real-time vision, we present an *approximately incremental* strategy for CNN execution. This strategy builds on ideas in incremental computation [7, 8]: it runs the full CNN for an initial input frame and records the input pixels and the output activation data. For subsequent frames, it detects changes in the input with respect to the saved frame and uses these changes to update the saved output. The incremental update is much cheaper than full CNN execution, so the performance and energy benefits outweigh the cost of detecting input changes. The updated output, however, need not match the full CNN execution exactly: instead, the incremental update is a close approximation of the original CNN layers. We apply ideas from approximate computing [9, 10, 11] to adaptively control the trade-off between vision accuracy and resource efficiency by falling back to full computation when the prediction error is likely to be high.

Our algorithm, *activation motion compensation* (AMC), takes inspiration from video codecs [12] and work on exploiting optical flow for computer vision [13, 14]. AMC captures visual motion in the input video and uses it to transform saved CNN activations. When pixels move in the input scene, AMC moves the corresponding values in the activation data. The algorithm skips a series of layers in the CNN by predicting their output and then invokes the remaining layers to compute the final vision result. We describe the mathematical relationship between pixel motion and convolutional layers and develop a new, hardware-optimized motion estimation algorithm to exploit this relationship.

We design a new hardware module, EVA^2, that implements the AMC algorithm. Instead of designing a CNN accelerator from scratch, we show how to apply EVA^2 to any generic CNN accelerator to improve its efficiency for live computer vision. EVA^2 adds new logic and memories to a baseline accelerator to skip the majority of CNN layer executions for the majority of frames. EVA^2 uses an adaptive control scheme to decide which frames to run precisely.

We implement EVA^2 and synthesize the design with the Synopsys ASIC toolchain, including the 2016 versions of Design Compiler, IC Compiler, and PrimeTime. To evaluate the system, we augment state-of-the-art designs for accelerating convolutional and fully-connected layers [2, 6] and find that EVA^2's makes up 3.6% of the full ASIC's area. On three CNN-based vision workloads, using EVA^2 reduces the energy cost per frame by 54.2%, 61.7% and 87.6% while also decreasing average frame latency by 54.0%, 61.7%, and 87.4% with less than 1% loss in vision accuracy.

2. ACTIVATION MOTION COMPENSATION

This section introduces activation motion compensation (AMC), our strategy for exploiting temporal redundancy in

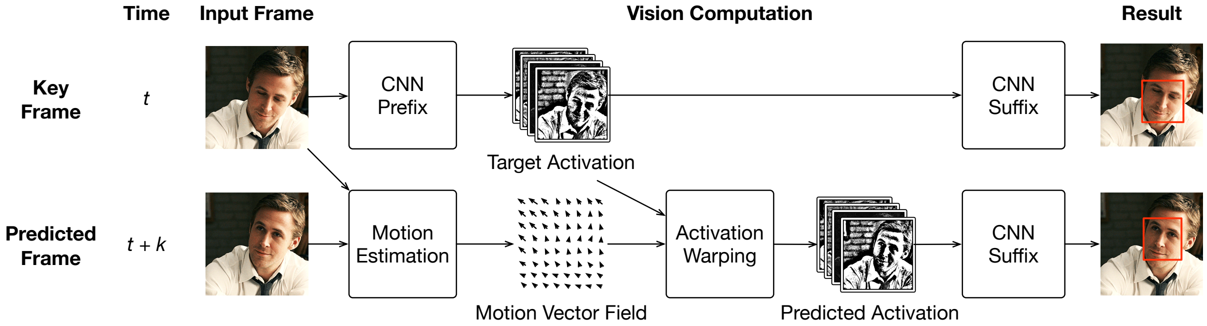


Figure 1: Activation motion compensation (AMC) runs the CNN precisely for periodic *key frames* and uses an approximately incremental update for more frequent *predicted frames*. For predicted frames, the algorithm estimates motion in the input and uses the resulting vector field to update a saved CNN activation from the last key frame.

CNNs for efficient real-time computer vision. The central idea is to use saved outputs from earlier *key frames* to predict the output for later *predicted frames*. As in traditional incremental computing [7, 8], the challenge is to find an algorithm for updating saved results that is much cheaper than running the entire computation "from scratch" on the new input. Unlike traditional strategies, however, our approach for vision is *approximately incremental:* the updated output need not match the original output values precisely. Instead, predicted frames need only yield equivalently high vision accuracy results.

One natural strategy for approximately incremental CNN execution would exploit the differentiability of neural network layers. If $f(x)$ is a layer, then there must be a derivative function df such that $f(x+dx) \approx f(x) + df(dx)$. *Delta networks* operate by storing the old activation, $f(x)$, for every layer, computing $df(dx)$ for new layers, and adding it to the stored data [15, 16]. While delta updates are straightforward to implement, they do not address the primary efficiency bottlenecks in CNN execution. First, the hardware must store the activation data for every network layer to apply per-layer deltas, exacerbating the already-large memory requirements for CNN execution. Second, to compute $df(dx)$ for every layer, the hardware must load the full set of model weights, and previous work shows that the cost of loading weights can dominate the computation cost in CNNs [17, 2]. Finally, using pixelwise derivatives to represent changes in video frames assumes that individual pixels change their color slowly over time. If a camera pans slightly or objects move in the scene, however, most pixels will change abruptly.

Instead of relying on a pixel-level derivative, our technique uses visual motion in the input scene. The intuition is the same as in video compression: most frames are approximately equal to the previous frame with some blocks of pixels moved around. AMC detects pixel motion and compensates for it in the output of a *single* target layer in the CNN. AMC builds on recent computer vision work to *warp* the target CNN activation data based on motion information [13]. Unlike delta updates, AMC bypasses the computation and memory accesses for an entire sequence of CNN layers.

2.1 AMC Overview

Figure 1 illustrates AMC's approximately incremental execution strategy for real-time vision using CNNs. AMC processes the input video as as a mixture of *key frames*, which undergo full and precise CNN execution, and *predicted frames*, which use cheaper, approximate execution. AMC saves intermediate results during key frames and incrementally updates them for predicted frames. Section 2.3.4 describes how to decide which frames should be key frames.

To apply the strategy to a new CNN architecture, the system splits the CNN's series of layers into two contiguous regions: a larger prefix that only executes for key frames, and a smaller suffix that executes on every frame. The final layer in the prefix is called the *target layer:* AMC saves the output from this layer during key frames and predicts its output during predicted frames. Intuitively, the convolutional and pooling layers in the prefix have a spatial relationship to the input, while the suffix contains fully-connected layers and other computations where scene motion can have unpredictable effects. The prefix typically performs more generic feature extraction, while later layers usually vary more between vision tasks [18]. Section 2.3.5 describes how to choose the target layer in more detail.

During key frames, AMC stores the network's input and the target layer's output for reuse during predicted frames. While the input image is typically small, CNN activations can be high-dimensional and occupy multiple megabytes [3]. Section 2.3.2 describes how to exploit activation sparsity to efficiently store key frame outputs.

During predicted frames, AMC detects the motion between the last stored key frame and the new input frame. *Motion estimation* is the problem of computing a vector field describing the visual displacement between two input frames. Motion estimation algorithms offer different trade-offs between granularity, accuracy, and efficiency [19, 20, 21, 22, 23, 24, 25]. AMC requires an efficient motion estimation technique that is aligned with the CNN's convolutional structure. Section 2.3.1 describes *receptive field block motion estimation* (RFBME), our new algorithm optimized for the hardware implementation of AMC.

Next, we describe mathematically how AMC updates stored activation data using motion information. Then, Section 2.3 fills in the details for a complete implementation of AMC.

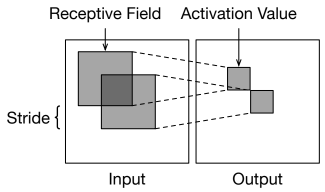


Figure 2: A convolutional layer applies filters to regions in the input. The input region corresponding to a given activation value is called its *receptive field*.

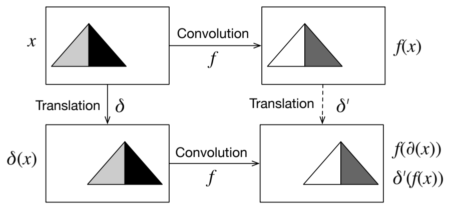


Figure 3: Convolutional layers and translations are *commutative*, so $f(\delta(x)) = \delta'(f(x))$ where δ' is a scaled version of the translation δ.

2.2 Warping CNN Activations

The core challenge in AMC is accurately updating saved activation data at the target layer according to input motion. *Activation warping*, as shown in Figure 1, takes in an old activation and a vector field and produces an updated activation. To perform warping, AMC needs to convert the vector field describing motion in the input image into a corresponding vector field for the target activation data. This conversion depends on the structure of convolutional layers, illustrated in Figure 2. Convolutional layers scan over the input using a fixed stride and apply the dot product of a filter matrix and a region in the input to produce each output value. This input region corresponding to each output value is called its *receptive field*. By propagating this structure through multiple convolutional layers, a receptive field in the input pixels can be defined for every activation value at every layer. Using receptive fields, we can derive the relationship between motion in the input image and in the target activation layer.

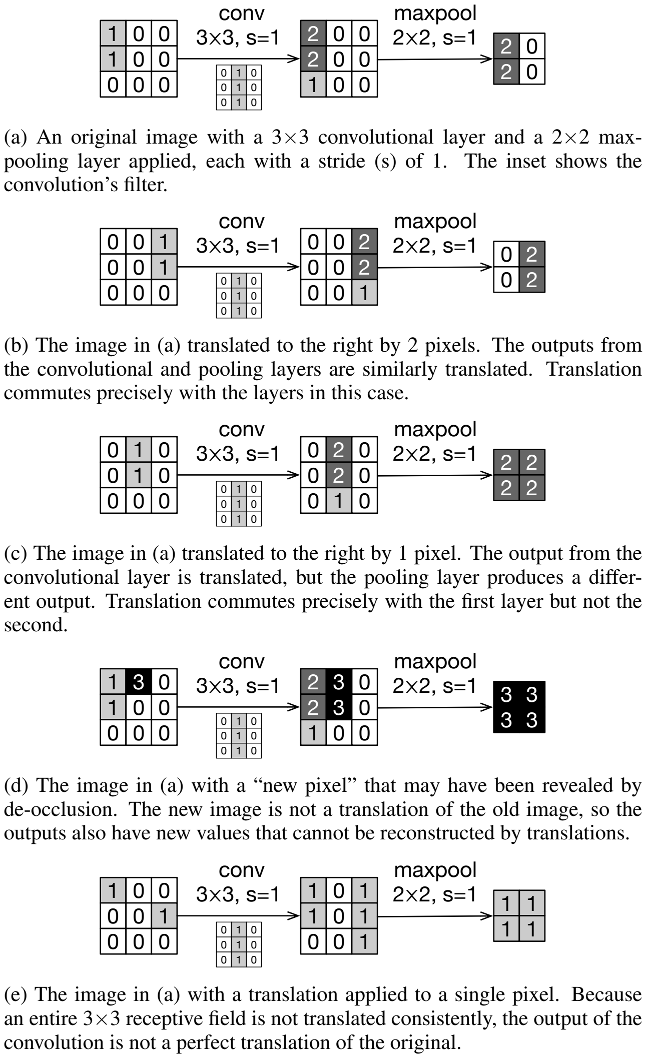
Convolutions and translations commute.

The key insight is that, when the pixels in a receptive field move, they cause their corresponding activation value in the target layer to move by a similar amount. Mathematically, convolutional layers *commute* with translation: translating (i.e., moving) pixels in the input and then applying a convolution is the same as applying the convolution to the original pixels and then translating its output. (See Figure 3.)

Let f be a convolutional layer in a CNN, let x be an input image, and let δ be the vector field describing a set of translations in x. We define $\delta(x)$ to be the new image produced by translating pixels in x according to the vector field δ. We can also produce a new vector field, δ', by scaling δ by the stride of the convolution: for a convolutional layer with stride s, a distance d in the input is equivalent to a distance $\frac{d}{s}$ in the output. The commutativity of translations and convolutions means that:

$$f(\delta(x)) = \delta'(f(x))$$

In AMC, x is a key frame and $f(x)$ is the target activation for that frame. If $\delta(x)$ is a subsequent input frame, AMC can use the saved output $f(x)$ and compute $\delta'(f(x))$ instead of running the full computation f on the new input.



(a) An original image with a 3×3 convolutional layer and a 2×2 max-pooling layer applied, each with a stride (s) of 1. The inset shows the convolution's filter.

(b) The image in (a) translated to the right by 2 pixels. The outputs from the convolutional and pooling layers are similarly translated. Translation commutes precisely with the layers in this case.

(c) The image in (a) translated to the right by 1 pixel. The output from the convolutional layer is translated, but the pooling layer produces a different output. Translation commutes precisely with the first layer but not the second.

(d) The image in (a) with a "new pixel" that may have been revealed by de-occlusion. The new image is not a translation of the old image, so the outputs also have new values that cannot be reconstructed by translations.

(e) The image in (a) with a translation applied to a single pixel. Because an entire 3×3 receptive field is not translated consistently, the output of the convolution is not a perfect translation of the original.

Figure 4: An example convolutional layer and pooling layer applied to transformations of an input image.

Figure 4 shows an example convolution. If Figure 4a is a key frame, then Figures 4b and 4c show translations of the key frame to the right by 2 and 1 pixels, respectively. In both cases, the output of the convolution is translated by the same amount in the same direction.

Sources of approximation.

In this mathematical model, activation warping is perfectly precise: an incremental update produces the same result as the full CNN computation. However, the formulation makes strong assumptions that do not generally hold for real video and complete CNNs. To the extent that these conditions are violated, AMC is approximate.

To clarify the sources of approximation in AMC, we list sufficient conditions under which AMC is precise. We demonstrate the conditions by example and describe how AMC mitigates—but does not eliminate—errors when the conditions are false. AMC relies on the inherent tolerance in deep neural networks to tolerate the remaining imprecision.

Condition 1: Perfect motion estimation. AMC is precise when motion estimation perfectly captures the changes in the input. In other words, given a key frame x_0 and a predicted frame x_1, motion estimation always finds a vector field δ such that $\delta(x_0) = x_1$ exactly. In reality, motion estimation algorithms are imperfect and not all changes between frames are attributable to motion.

For example, lighting changes and de-occlusion cause "new pixels" that are not present in the previous frame. Figure 4d shows an modified version of the input in Figure 4a with a new pixel. The output of the convolution, correspondingly, is not a translation of the original output.

Condition 2: Convolution-aligned motion. To perfectly translate the pixel vector field δ into an activation vector field δ', AMC assumes that blocks of pixels in receptive fields move in increments according to the convolution's stride. If a vector in δ has magnitude d, the corresponding vector in δ' has magnitude $\frac{d}{s}$ where s is the convolutional layer's stride. If d is not a multiple of s, the translation ends at a fractional coordinate in the activation and perfect reconstruction is impossible. Similarly, if pixels within a single receptive field move differently, a translation in the activation cannot perfectly capture the change.

Figure 4e, for example, shows a version of Figure 4a with a single pixel translated. Because the translation's granularity is smaller than a receptive field, the output of the convolution is not a perfect translation of its original output.

Condition 3: Nonlinearities preserve motion. Translation is commutative with convolution, but CNNs have other kinds of layers, such as pooling, that are not perfectly commutative. For example, Figures 4b and 4c show translated versions of the input in Figure 4a. In the first translation, a max-pooling layer produces and output that is a perfect translation of the original output. In the second, however, the max-pooling layer creates a non-translated output.

Real video and real CNNs violate all three of these conditions, so AMC's activation warping is an approximation of plain CNN execution. The central challenge in the design of the rest of the AMC algorithm is avoiding and suppressing these potential errors. AMC's adaptive key frame selection (Section 2.3.4) mitigates motion estimation errors by falling back to precise execution; interpolated warping (Section 2.3.3) addresses unaligned motion; and careful target layer selection (Section 2.3.5) avoids too much error accumulation due to nonlinear layers.

2.3 Design Decisions for AMC

This section outlines our approaches to motion estimation, activation storage, interpolation, key frame selection, and target layer selection. AMC is a design space: for each of these factors, there is a range of possible choices. We describe both the general requirements and the specific tactic we use in our hardware implementation.

2.3.1 *Efficient Motion Estimation*

The first step in executing an AMC predicted frame is motion estimation. Motion estimation takes two input frames and produces a 2D vector field describing the visual displacement between the frames. There are many motion estimation algorithms in the literature, including block matching [19, 20], phase correlation in the Fourier domain [21], traditional optical flow algorithms such as Lucas–Kanade [22] and Horn–Schunck [23], and deep learning methods [24, 25]. A primary difference is the motion granularity: optical flow algorithms produce dense, pixel-level vector fields, while block matching detects coarser motion but can often be cheaper. Previous work on exploiting motion for efficient vision has used pixel-level optical flow [13]. For AMC, however, pixel-level motion estimation yields unnecessary detail: activation warping can only handle motion at the granularity of entire receptive fields.

Block matching algorithms, often used in video codecs, work by taking a block of pixels and comparing it to a window of nearby blocks in the reference (key) frame. The location of the closest matching reference block determines the motion vector for the block. Critical parameters include choice of search window, organization of search within this window, and similarity metric for two candidate blocks [12].

We develop a block matching algorithm, *receptive field block motion estimation* (RFBME), that is specialized for AMC. RFBME estimates the motion of entire receptive fields. The resulting displacement vector for each input receptive field maps to a corresponding displacement vector for a value in the target activation. RFBME works by dividing the input into square *tiles* whose size is equal to the receptive field stride. It first compares individual tiles to each other, computes their l_1 norm difference, and then sums the differences for the tiles that make up each receptive field. This two-step process enables reuse between different pairs of receptive fields. Section 3.1 describes the algorithm in detail and its hardware implementation.

2.3.2 *Efficient Activation Storage*

AMC needs to store the target activation data for the key frame so that subsequent predicted frames can reuse it. Activations for CNNs, however, can be large: storing an activation naively would require multiple megabytes of on-chip memory. To mitigate storage costs, we exploit *sparsity* in CNN activations. Many CNN accelerator proposals also exploit sparsity: most values in CNN weights and activations are close to zero, so they can be safely ignored without a significant impact on output accuracy [2, 3, 4, 6]. We use the same property to avoid storing near-zero values. Section 3.2 describes our hardware design for encoding and decoding sparse data.

2.3.3 *Interpolated Warping*

AMC's activation warping step takes a vector field δ and an old CNN activation and updates the activation according to the translations in the vector field δ'. It works by scaling the vector field, which was produced for the input dimensions, to match the dimensions of the activation data. This scaling can result in vector distances that are not integer values of pixels. To translate by a fractional distance, AMC needs to interpolate the values of nearby activations.

There are a range of possible strategies for interpolation, including cheap techniques, such as nearest neighbor and bilinear interpolation, to more computationally expensive but accurate interpolation methods that preserve edge or gradient boundaries. For this paper, we choose bilinear interpolation to average neighboring pixels in 2D space while still maintaining high performance. Our experiments suggest that bilinear interpolation improves vision accuracy by 1–2% over simple nearest neighbor matching on average for one CNN benchmark (FasterM).

2.3.4 *Selecting Key Frames*

The primary control that AMC has over vision accuracy and execution efficiency is the allocation of key frames, which are both more expensive and more accurate than predicted frames. A broad range of strategies exist to decide when to use each type of frame. The simplest is a static key frame rate: every *n*th frame is a key frame, and the rest are predicted frames. A adaptive strategy, however, can allocate more key frames when the scene is chaotic and unpredictable and fewer when AMC's predictions are more likely to succeed. To implement a adaptive strategy, the accelerator needs to measure some feature of the input scene that correlates with the probability of a successful AMC prediction. We consider two possible features:

- *Motion error.* AMC can produce a poor prediction when the motion estimation fails to accurately reflect the changes in the scene. To measure motion accuracy, we can reuse the internal bookkeeping of a block matching algorithm, which must compute the match error for each block in the scene. When the aggregate error across all blocks is high, this strategy allocates a new key frame. When large occlusions occur, for example, this method will identify the inadequacy of the motion information.

- *Total motion magnitude.* AMC's predictions are more accurate when there is less motion in the input scene. To measure the amount of motion, this simple strategy sums the magnitude of the vectors produced by motion estimation. This policy uses a key frame when the total amount of motion is large.

We implement and measure both techniques to compare their effectiveness. Section 4.4.5 quantitatively compares their effects on overall accuracy.

2.3.5 *Choosing the Target Layer*

To apply AMC to a given CNN, the system needs to choose a target layer. This choice controls both AMC's potential efficiency benefits and its error rate. A later target layer lets AMC skip more computation during predicted frames, but a larger CNN prefix can also compound the influence of layers that make activation warping imperfect. Some kinds of layers, including fully-connected layers, make activation warping impossible: they have no 2D spatial structure and no meaningful relationship with motion in the input. These *non-spatial* layers must remain in the CNN suffix, after the target layer. Fortunately, these non-spatial layers are typically located later in CNNs, in the more task-specific segment of the network [18]. Our proposal also cannot predict layers within network structures based on an internal memory state, such as recurrent neural networks and LSTMs [26]. However, even these specialized networks tend to contain early sets of convolutional layers for feature extraction [27, 28], where AMC can apply.

As with key frame selection, the system can use a static or adaptive policy to select the target layer. In our evaluation, we measure the impact of choosing earlier and later CNN layers, up to the last spatial layer (Section 4.4.3) and find that the accuracy difference is negligible. Therefore, we implement AMC by statically targeting the last spatial layer. Dynamic policies for choosing the target layer may be useful future work if more complex networks demonstrate meaningful differences in accuracy for different input frames.

2.4 Intuition Behind AMC's Efficiency Advantage

While predicted frames obviate the need for CNN prefix computation, they do require motion estimation and motion compensation. To enhance intuition for why this is a desireable tradeoff, we demonstrate a first-order complexity evaluation. CNN prefix computation is dominated by the multiply-and-accumulates (MACs) in the convolutional layers, and predicted frame computation is dominated by RFBME's pixel adds. Total prefix MACs are given by:

$$
\begin{aligned}
\text{\# of outputs} &= \text{layer width} \times \text{layer height} \times \text{out channel}, \\
\begin{array}{r}\text{\# of MACs}\\ \text{per output}\end{array} &= \text{in chan} \times \text{kernel height} \times \text{kernel width}, \\
\begin{array}{r}\text{\# of layer}\\ \text{MACs}\end{array} &= \text{\# outputs} \times \text{\# of MACS per output}, \\
\begin{array}{r}\text{\# of prefix}\\ \text{MACs}\end{array} &= \sum_{i=1}^{\text{num layers}} \text{\# of MACs for all layers}[i].
\end{aligned}
$$

For a Faster16 prefix ending at layer conv5_3 on 1000×562 images, for example, the total is 1.7×10^{11}. In contrast, the total number of additions in RFBME is given by:

$$
\begin{aligned}
\begin{array}{r}\text{unoptimized}\\ \text{RFBME ops}\end{array} &= (\text{layer width} \times \text{layer height}) \\
&\quad \times \left(\frac{2 \times \text{search radius}}{\text{search stride}}\right)^2 \times \text{rfield size}^2, \\
\text{tiled RFBME ops} &= \frac{\text{unoptimized RFBME ops}}{\text{rfield stride}^2} \\
&\quad + \left(\frac{\text{rfield size}}{\text{rfield stride}}\right)^2.
\end{aligned}
$$

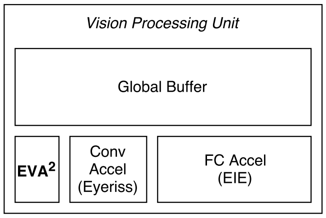


Figure 5: EVA^2 as a part of a complete vision processing unit (VPU). CNNs consist primarily of convolutional layers and fully-connected layers; this example VPU uses ASIC designs from the architecture literature for each of the two layer types [2, 6] and adds EVA^2.

For Faster16:

$$\begin{aligned}\text{unoptimized RFBME ops} &= (63 \times 36) \times \left(\frac{2 \times 50}{16}\right)^2 \times (196)^2 \\ &\approx 3 \times 10^9, \\ \text{tiled RFBME ops} &= \frac{3,403,417,500}{16^2} + \left(\frac{196}{16}\right)^2 \\ &\approx 1.3 \times 10^7.\end{aligned}$$

Hence, AMC eliminates $\sim 10^{11}$ MACs and incurs $\sim 10^7$ additions for RFBME in this case.

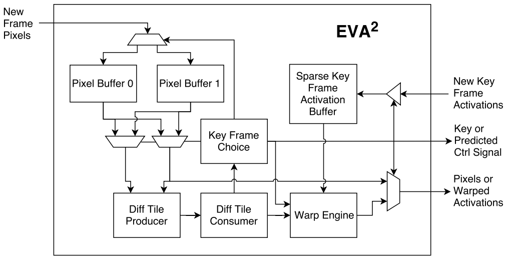

New Frame Pixels
EVA^2
Pixel Buffer 0
Pixel Buffer 1
Sparse Key Frame Activation Buffer
New Key Frame Activations
Key Frame Choice
Key or Predicted Ctrl Signal
Pixels or Warped Activations
Diff Tile Producer
Diff Tile Consumer
Warp Engine

Figure 6: The architecture of EVA^2.

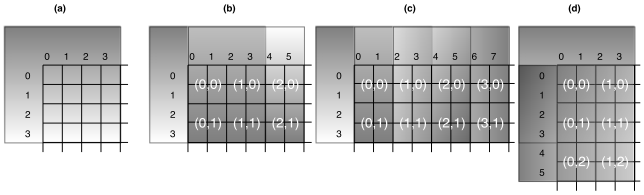


Figure 7: A set of example receptive fields with size 6, stride 2, and padding 2. Nearby receptive fields overlap significantly, and receptive fields near the edge overlap enclose out-of-bounds pixels.

3. THE EMBEDDED VISION ACCELERATOR ACCELERATOR

This section describes the design of the Embedded Vision Accelerator Accelerator (EVA^2): our efficient hardware implementation of the AMC algorithm. EVA^2 is not a complete CNN accelerator. Instead, we design it to complement existing deep learning ASIC designs. Figure 5 shows how EVA^2 fits into a complete vision processing unit (VPU) that also includes hardware for executing the convolutional and fully-connected layers that make up the bulk of CNN computation. EVA^2 makes no assumptions about layer computation, so only two minor changes are required for any given CNN accelerator: 1) multiplexing to receive input from off-chip (for key frames) or from EVA^2 (for predicted frames), and 2) recoding logic to convert between the CNN accelerator's encoding for activations and EVA^2's run-length activation encoding. EVA^2 communicates with the layer accelerators through a global buffer or on-chip bus.

When the VPU starts processing a new frame, EVA^2 performs motion estimation and decides whether to run the computation as a key frame or a predicted frame. For key frames, EVA^2 sends the unmodified pixels to the layer accelerators and invokes them to run the full CNN. For predicted frames, EVA^2 instead performs activation warping and invokes the layer accelerators to compute the CNN suffix.

Figure 6 shows the high-level architecture of EVA^2. Two *pixel buffers* store video frames: one pixel buffer holds the most recent key frame and the other holds the current input frame. The *diff tile producer* and *diff tile consumer* cooperate to run motion estimation. The *key frame choice* module uses absolute pixel differences from the diff tile consumer to decide the next key frame. For predicted frames, the diff tile consumer also sends a motion vector field to the *warp engine*, which updates the buffered key frame activation and sends the warped activation to the layer accelerators. For key frames, the layer choice model toggles the muxing for the pixel buffers to reverse their roles and sends the pixels to the layer accelerators. When the layer accelerators send the target activation data back, EVA^2 stores it in its sparse key frame activation buffer.

The rest of this section describes the design of EVA^2's two main logic components: the motion estimation logic, consisting of a diff tile producer and consumer, and the motion compensation logic, consisting of the warp engine.

3.1 Receptive Field Block Motion Estimation

EVA^2 implements receptive field block motion estimation (RFBME), the specialized block matching algorithm described in Section 2.3.1. The algorithm exploits two properties of receptive fields for efficiency. First, the receptive field size is typically much larger than its stride, so nearby receptive fields overlap significantly and can reuse computation. Second, layer padding means that receptive fields often exceed the bounds of the image, and comparisons with these out-of-bounds pixels are unnecessary.

Figure 7 shows an example input matrix where the receptive fields have size 6, stride 2, and padding 2. The first receptive field (a) extends beyond the image bounds, so only 16 of its 36 pixels are valid. The second receptive field (b) entirely encloses the first receptive field's valid pixels, and the third (c) overlaps partially, in the range $x \in [2,3], y \in [0,3]$. The receptive fields in this example overlap on 2×2 *tiles*. In general, these tiles are $s \times s$ squares where s is the receptive field's stride. RFBME divides the image and the receptive fields into tiles for comparison. When the receptive field size is not an integer multiple of the stride, RFBME ignores partial tiles; we find that these border pixels do not have a significant impact on RFBME's motion vectors.

The key insight is that the total absolute pixel difference for a receptive field is the sum of the differences of its tiles, and these tile differences are shared between many receptive fields. For example, imagine that the algorithm has already computed tile-level differences for all four tiles in the first receptive field, shown in Figure 7(a), at a given offset. The second receptive field (b) shares these same four tiles and adds two more: the tiles labeled $(2,0)$ and $(2,1)$. To compute a receptive field difference at the same offset, the algorithm can reuse the previously-computed differences for the first four tiles and add on differences for the two new tiles. The potential benefit from this reuse depends linearly on the number of pixels per tile. While the stride in this example is small, the stride in later layers of modern CNNs can be 16 or 32 pixels, which exponentially increases the amount of shared pixels per tile.

To exploit this reuse of tile differences, our RFBME microarchitecture uses a producer–consumer strategy. The *diff tile producer* compares $s \times s$ tiles to produce tile-level differences, and the *diff tile consumer* aggregates these differences to compute receptive field differences. The consumer then finds the minimum difference for each receptive field; this determines its offset and the output of RFBME. The next two sections describe each stage in detail.

3.1.1 Diff Tile Producer

For each tile, the diff tile producer performs a search across the key frame according to a fixed *search stride* and *search radius*. It uses a subsampled traditional exhaustive block matching search [12]. This search considers all locations in the key frame that are aligned with the search stride and are within the search radius from the tile's origin. A wider radius and a smaller stride yield higher accuracy at the expense of more computation.

To perform the search, the diff tile producer first loads the current tile from the pixel buffer that contains the predicted frame. Then, it iterates over the valid (in-bounds) search offsets in the key frame according to the search parameters. For each valid offset, the producer computes an absolute pixel difference with the current tile using an adder tree. When the search is complete, the producer moves on to the next tile in the predicted frame and starts the search again.

3.1.2 Diff Tile Consumer

The diff tile consumer receives tile differences from the producer and coalesces them into full receptive field differences. Figure 8 shows its architecture including memories for caching re-used tiles and partial sums, as well as the pipelined adder trees for incremental updates. At a high level, the consumer runs a sliding window, the size of a receptive field, across a frame while summing the tile differences within the receptive field.

The consumer receives tile differences, streaming in row by row, from the producer and stores them in a *tile memory*. It buffers the incoming differences until it receives all the tiles for a receptive field. For example, in Figure 7(a), the first receptive field difference will be calculated once tile difference for $(1,1)$ in the second row has been received.

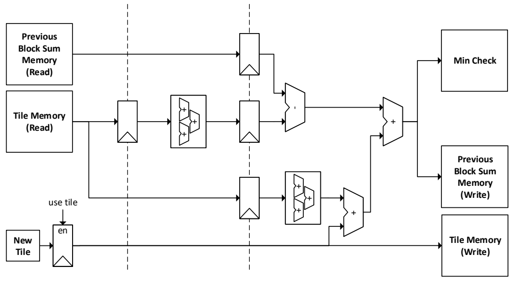

Previous Block Sum Memory (Read)
Tile Memory (Read)
use tile
en
New Tile
Min Check
Previous Block Sum Memory (Write)
Tile Memory (Write)

Figure 8: The architecture of the diff tile consumer used for receptive field block motion estimation (RFBME).

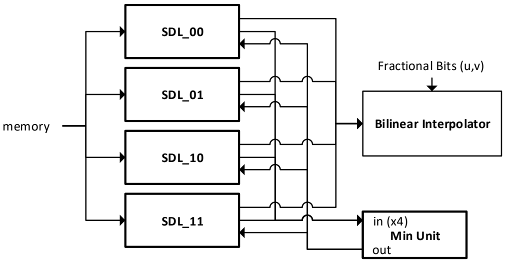


Figure 9: The architecture of the warp engine.

To enable computation reuse, the consumer stores each new receptive field difference it computes in a *past-sum* memory. To compute a new receptive field difference, it loads the difference for the previous overlapping receptive field, adds a new column tile differences, and subtracts the old column of tile differences. For example, in Figure 7(c), the second receptive field difference is the sum of the differences for the tiles $(0,0)$ through $(2,1)$. To calculate the difference for the third receptive field, the consumer fetches this value from the past-sum memory, adds the new tile column $(3,0)$ and $(3,1)$, and subtracts the old tile column $(0,0)$ and $(0,1)$. These rolling additions and subtractions avoid the need for exhaustive sums in the steady state.

The consumer checks every new receptive field difference against a *min-check* memory to find the minimum difference. When it finds a new minimum value, the consumer writes the difference and offset back to the min-check memory.

When the consumer finally finishes processing all the receptive fields, it sends the minimum-difference offsets, which constitute motion vectors, to the warp engine. It also sends the minimum differences themselves to the key frame choice module, which uses the total to assess the aggregate quality of the block matching.

3.2 Warp Engine

Figure 9 shows the warp engine's architecture, consisting of four sparsity decoder lanes, a bilinear interpolator, and a min unit. Each receptive field vector produced by the diff tile consumer ends at a fractional destination between four neighboring activation values. For each incoming vector,

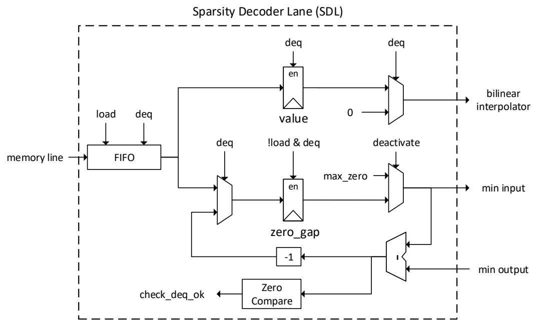


Figure 10: The warp engine's datapath for loading sparse activation data from the key activation memory.

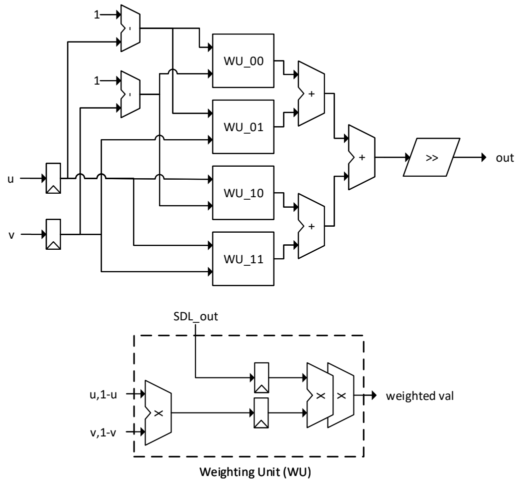


Figure 11: The warp engine's bilinear interpolation logic.

the warp engine loads this neighborhood of activation values from its sparse memory and feeds them into the bilinear interpolator, as well as the fractional bits of this motion vector. When interpolation for all vectors is complete, the warp engine sends the warped activations to the CNN accelerator to compute the CNN suffix.

To begin, the warp engine loads in activation data into its sparsity decoder lanes. Because CNN activations are inherently sparse, especially for later layers, we implement run-length encoding for activation storage to reduce our memory footprint. For example, sparse encoding save more than 80% of the space required to store activations for VGG-16, including encoding overhead. This allows EVA^2 to compress activations so that they entirely fit in its 300kB on-chip buffer, even at the high resolutions required for object detection. The use of run-length encoding, however, complicates the task of loading individual values within the warp engine.

Rather than expanding the encoding out directly, the warp engine uses four *sparsity decoder lanes* (Figure 10) to skip zeros shared among all of the four activations used in interpolation. Once activations are loaded into the FIFO, each lane sends its zero gap for its first activation. The min unit checks the zero gap in each lane and sends the minimum to all of the lanes. Each lane then decrements its zero gap by this amount, thereby jumping all of the lanes forward at once in the channel space. All lanes with zero gaps of zero after the min subtraction then provide their value register as input to the bilinear interpolator. Lanes with positive zero gaps provide zero as the input to the interpolator.

The warp engine feeds activation outputs from the sparsity decoder lanes into the four weighting units in the bilinear interpolator (shown in Figure 11). The interpolator is a two-stage datapath which computes a 4-way weighted sum for each activation in the sparsity decoder lanes, SDL_00, SDL_01, SDL_10, and SDL_11, and the fractional bits of a given motion vector, (u, v). The computed weighted sum is:

$$\begin{gathered}(\text{SDL\_00} \times (1-u) \times (1-v)) + \\ (\text{SDL\_01} \times (1-u) \times v) + \\ (\text{SDL\_10} \times u \times (1-v)) + \\ (\text{SDL\_11} \times u \times v)\end{gathered}$$

Instead of rounding the 16-bit fixed-point values after each multiplication and sum, the results are shifted back to 16 bits once the final interpolated value is produced. Once all activations have been produced for the target warped layer the CNN accelerators can begin the CNN suffix computation.

4. EVALUATION

This section studies EVA^2's impacts on vision accuracy and efficiency and explores its implementation trade-offs.

4.1 Experimental Setup

We implement EVA^2 in RTL and synthesize our design using the Synopsys toolchain, including the 2016 versions of Design Compiler, IC Compiler, and PrimeTime. We use the TSMC 90 nm process library and scale the area, time, and power to match the 65 nm results for Eyeriss. For energy and timing, we run simulations of the full EVA^2 design. The EVA^2 implementation uses two eDRAM memories for its two main buffers, the pixel buffers and the activation buffer. We use CACTI 6.5 [29] to measure the power, performance, and area of the memories. CACTI includes both an eDRAM model and an SRAM model, which we use for EVA^2's smaller buffers.

Baseline accelerator.

We apply EVA^2 to a model of a state-of-the-art deep learning accelerator based on recent architecture papers. We model Eyeriss [2] for convolutional layers and EIE [6] for fully-connected layers by gathering published per-layer results from each paper. A journal paper about Eyeriss [30] reports both power and latency numbers for each layer in the VGG-16 [31] and AlexNet [32] networks; the EIE results only include latency numbers for these two networks and use total design power to estimate energy. Because Eyeriss is implemented on a TSMC 65 nm process and EIE on TSMC 45 nm, we normalize by scaling up the power, latency, and area for EIE according to the technology scaling factor. The total latency and energy for a CNN execution in our model is the sum of the individual layer costs. To quantify the cost of layers not present in AlexNet and VGG-16, the model scales the average layer costs based on the number of multiply–accumulate

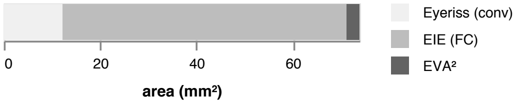


Figure 12: Hardware area on a 65 nm process for EVA^2 compared to deep learning ASICs: Eyeriss [2] for convolutional layers and EIE [6] for fully-connected layers.

operations required for each layer, which we find to correlate closely with cost in both accelerators [33].

EVA^2 hardware.

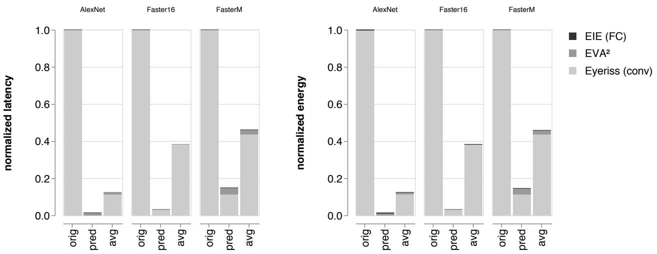
Our RTL implementation of EVA^2 meets timing with a clock cycle of 7 ns. Figure 12 compares the area for EVA^2 against the reported area for the convolutional layer accelerator (Eyeriss) and the fully-connected accelerator (EIE). The area for Eyeriss [2] is 12.2 mm^2 on a 65 nm process, 78.6% of which is occupied by its PEs. The area for EIE [6] is 40.8 mm^2 on a 45 nm process; compensating for the process difference, EIE would occupy approximately 58.9 mm^2 on a 65 nm process. EVA^2 itself occupies 2.7 mm^2, which is 3.6% of the overall area for the three units. Of this, the eDRAM memory for the pixel buffers occupies 53.1% of EVA^2's area, and the activation buffer occupies 15.6%.

Vision applications and dataset.

We study three convolutional neural networks. **AlexNet** [32] is an object classification CNN consisting of 5 convolutional layers and 3 fully-connected layers. **Faster16** is an version of the Faster R-CNN object detection network [34], which can use different networks for its feature extraction phase, based on the VGG-16 recognition network [31]. VGG-16 has 16 convolutional layers; Faster R-CNN adds 3 convolutional layers and 4 fully-connected layers. **FasterM** is a different variant of Faster R-CNN based on the "medium" CNN-M design from Chatfield et al. [35]. CNN-M has only 5 convolutional layers, so it is smaller and faster but less accurate than VGG-16. We use standard vision metrics to assess EVA^2's impact on accuracy. For our classification network, AlexNet, we use top-1 accuracy; for the object detection networks, we use mean average precision (mAP).

To train, validate, and test the CNNs we use the Google YouTube-BoundingBoxes dataset (YTBB) [36]. This dataset consists of 240,000 videos annotated with object locations and categories. It includes ground truth for both object detection and frame classification. Training, testing, and reference frames were all decoded at 30 frames per second, corresponding to a 33 ms time gap between each frame. The total dataset is more than five times larger than ImageNet [37], so we use subsets to reduce the time for training and testing. We train on the first 1⁄25 of the training datasets (132,564 and 273,121 frames for detection and classification, respectively). To evaluate during development, we used a validation dataset consisting of the first 1⁄25 of each of YTBB's validation sets (17,849 and 34,024 frames for the two tasks). For the final results reported in this section, we use a fresh test set consisting of the last 1⁄25 of YTBB's validation sets.

Training hyperparameters were used without modification from open-source Caffe [38] implementations of each network. All three networks were initialized with weights pretrained on ImageNet [37]. pyCaffe was used to train on a Xeon server using two NVIDIA GTX Titan X Pascal GPUs.



(a) Latency per frame.

(b) Energy cost per frame.

Figure 13: Performance (a) and energy (b) impact of EVA^2 compared to a baseline CNN accelerator. *Orig* shows the unmodified CNN execution; *pred* shows the cost of predicted frames with EVA^2; and *avg* shows the overall average cost per frame.

4.2 Energy and Performance

Figure 13 depicts the energy savings that EVA^2 offers over the baseline CNN accelerator. The figure shows the energy and latency cost for processing a single frame on the baseline accelerator without EVA^2, the average with EVA^2 enabled, and the costs for EVA^2's key frames and predicted frames individually. In these configurations, the degradation on the application's vision quality score is at most 1 percentage point. Even the smallest savings are significant. For FasterM, the energy cost with EVA^2 is 46.0% of the baseline cost. The savings are particularly dramatic for AlexNet because EVA^2 adapts to an extremely low key frame rate in the classification setting; the next section describes this effect in more detail.

Both the energy and latency of the fully-connected layers are orders of magnitude smaller than the corresponding costs for convolutional layers. This difference is due to EIE's efficiency: it exploits the redundancy in fully-connected layer weights and can store the entire model on chip [6]. This on-chip storage is reflected in EIE's low latency and energy and its large area requirement (see Figure 12). Eyeriss's use of off-chip DRAM is representative of other CNN ASIC designs [39, 2, 3].

4.3 Accuracy–Efficiency Trade-Offs

To quantify AMC's trade-off space between efficiency and accuracy, we consider three configurations with different key frame rates. Table 1 lists the accuracy and efficiency impact of three configurations, *hi*, *med*, and *lo*, found by limiting the validation accuracy to $< 0.5\%$, $< 1\%$, and $< 2\%$, respectively. We measure the average frame cost and accuracy on a test set. The *med* configuration is also shown in Figure 13.

The measured accuracy drop in every configuration is small, and EVA^2's benefits improve as the accuracy constraint is loosened. For FasterM's *lo* configuration, for example, only 29.0% of the frames are key frames, so the average energy cost per frame is only 39.3% of the baseline, but the test accuracy drop remains less than 1%.

For AlexNet, extremely low key frame rates suffice. Even

Network	Config	Acc.	Keys	Time (ms)	Energy (mJ)
AlexNet	orig	65.1	100%	115.4	32.2
	hi	65.1	22%	26.7	7.4
	med	64.3	11%	14.5	4.0
	lo	63.8	4%	5.9	1.6
Faster16	orig	60.1	100%	4370.1	1035.5
	hi	60.0	60%	2664.8	631.3
	med	59.4	36%	1673.6	396.4
	lo	58.9	29%	1352.7	320.3
FasterM	orig	51.9	100%	492.3	116.7
	hi	51.6	61%	327.2	77.4
	med	51.3	37%	226.4	53.5
	lo	50.4	29%	194.7	45.9

Table 1: The trade-off space between accuracy and resource efficiency with EVA^2. For the original baseline and three key configurations, we show the vision task accuracy score (*acc*), the fraction of frames that are key frames (*keys*), and the average latency and energy cost per frame.

in the *hi* configuration, which has no accuracy drop within three significant figures, only 22.1% of frames are key frames. The reason is that, unlike object detection results, frame classification results change slowly over time. EVA^2's adaptive key frame selection can help decide when a new class may be likely, but the vast majority of frames have the same class as the previous frame.

4.4 Design Choices

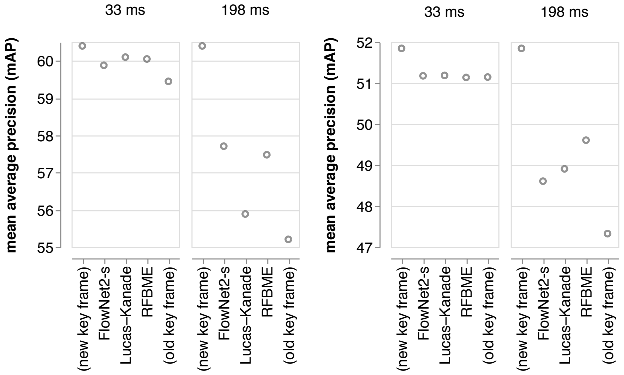
We characterize the impact of various design choices in EVA^2's implementation of AMC.

4.4.1 Motion Compensation vs. Memoization

The first choice when using AMC for an application is if the system should simply memoize the key frame result and reproduce it for future predicted frames, or if each predicted frame should be produced through activation motion compensation and computing the subsequent CNN suffix. Applications such as object detection, semantic segmentation, and pose estimation are very sensitive to pixel translation. For these applications, motion compensation achieves improves accuracy over memoization (i.e., the old key frame). This includes Faster16 and FasterM as they are object detection networks, and Figure 14 shows their benefit from motion compensation. On the other hand, applications such as image classification (including the AlexNet network) are designed to be as translation insensitive as possible, meaning that motion compensation can only hurt performance by introducing noise. AlexNet's results for memoization for 4891 ms are only a percentage point worse than if the full CNN was used, however when using motion compensation the degradation is closer to 5%. For this reason, the experiments in this section run AlexNet with memoization and Faster16 and FasterM with motion compensation.

4.4.2 Motion Estimation

Motion estimation is a key ingredient in the AMC algorithm. We compare our custom block-based motion estimation algorithm, RFBME, with two pixel-level optical flow techniques: the classic Lucas–Kanade algorithm [22] and the FlowNet 2.0 CNN [25]. Unlike RFBME, both algorithms produce pixel-level vector fields. To convert a pixel vector field to a receptive field vector field we take the average vector within each receptive field.



(a) Faster16.

(b) FasterM.

Figure 14: Accuracy impact of motion estimation techniques. *New key frame* shows the ideal baseline accuracy when computing the full CNN precisely, *old key frame* shows the worst-case accuracy for using the previous key frame without updating it at all, and the rest are motion estimation algorithms. RFBME is our new algorithm.

Figure 14 shows the overall mean average precision for predicted frames in FasterM and Faster16 when using each method. We show the error when predicting at two different time intervals from the key frame: 33 ms, which is the next frame of video (where we expect the smallest possible amount of motion relative to the tested frame), and 198 ms which was chosen as it consistently resulted in accuracy degredation for our translation sensitive benchmarks which could be recovered via motion compensation. In all cases for our object detection benchmarks, RFBME yields either the best or nearly the best accuracy. Because it does not come at a disadvantage in vision accuracy, and both FlowNet and Lucas–Kanade's per-pixel computation may be more expensive than RFBME's hardware-optimized data flow, we choose RFBME for our final EVA^2 design. FlowNet is particularly expensive, as the lowest-complexity "FlowNet2-s" [25] requires 12×10^9 MACs: orders of magnitude more computation than RFBME.

4.4.3 Target Layer

The choice of the *target layer* for AMC controls the amount of savings it can offer and its error. We study the accuracy impact of selecting an early target layer and a late target layer for each application. In each case, the early layer is after the CNN's first pooling layer, and the late layer is the last *spatial* layer: the layer before the first fully-connected layer or other computation would prevent activation warping. Table 2 shows the accuracy for AMC predicted frames when selecting each of these layers as the target layer. In most cases, the accuracy for predicting at the later layer is *higher* than for the earlier layer. The exception is Faster16 at 33 ms, where the difference is small. This improvement suggests that AMC's activation updates are accurate enough to suffice even for a large CNN prefix. We use the later layer

Network	Interval	Early Target	Late Target
AlexNet	orig	63.52	63.52
	4891 ms	49.95	53.64
Faster16	orig	60.4	60.4
	33 ms	60.29	60.05
	198 ms	55.44	57.48
FasterM	orig	51.85	51.85
	33 ms	50.90	51.14
	198 ms	48.77	49.61

Table 2: The accuracy impact of targeting different layers for EVA^2's prediction at various key frame intervals. The *orig* rows show the baseline accuracy for each network for reference.

Network	Target Layer	Accuracy
FasterM	No Retraining	51.02
	Early Target	45.35
	Late Target	47.82
Faster16	No Retraining	60.4
	Early Target	61.30
	Late Target	60.52

Table 3: The accuracy impact of training to fine-tune CNNs for execution on warped data. The accuracy column shows the network's score when processing plain, unwarped activation data.

for each application in the rest of this evaluation.

4.4.4 *Training on Warped Activation Data*

While EVA^2's predictions approximate "true" activation data, it may introduce artifacts that interfere with the normal operation of the CNN suffix. To counteract these artifacts, we can retrain the CNN suffix on warped activation data. Table 3 examines the impact of this retraining on our FasterM and Faster16 by measuring the resulting accuracy on plain (key) frames. For Faster16, the impact of retraining is small for the early target layer and negligible for the later target layer. For FasterM, retraining actually *decreases* accuracy on key frames, although this may be due to it's limited training schedule in comparison with Faster16. We conclude that additional training on warped data is unnecessary.

4.4.5 *Key Frame Selection*

AMC can choose adaptively when to use expensive key frames and when to use cheap predicted frames. We evaluate two strategies that choose the frame mode by measuring a feature of the input data and applying a threshold. The first strategy uses the match error values produced during RFBME execution, which is high when our motion estimation algorithm fails to capture all the changes in the input. The second takes the total magnitude of the motion vector field, which is high when many pixels in the scene have large displacements.

Figure 15 compares the vision accuracy for each strategy. In these experiments, we fix a key frame interval and sweep the decision threshold, then measure the resulting fraction of predicted frames and the output accuracy. For a fair comparison between the two metrics, each data point contains the same percentage of predicted frames and key frames per each metric. A fixed key frame rate can be thought of as a straight line from the 0% predicted frames point to the 100%. As both of these metrics contain a significant amount of area above this random line, they are both viable adaptive key frame choice metrics. The results show that the two strategies also yield comparable accuracy. We use the block error metric in our hardware implementation because it is cheap to compute: block errors are a "byproduct" of RFBME.

5. RELATED WORK

EVA^2 builds on a wave of recent architectural work on efficient hardware for deep learning. For a survey of the state of the art, see the tutorial by Sze et al. [40]. Commercial hardware efforts include GPUs and manycores with customized vector and low-precision instructions [41, 42]. In research, many recent ASICs target convolutional and fully-connected layers [43, 17, 1, 44, 2, 45, 46, 39], as do some FPGA designs [47, 48, 49, 50]. Recently, accelerators have focused on exploiting sparsity in model weights and activations [6, 4, 3, 51] and extreme quantization to ternary or binary values [52, 53, 54]. EVA^2's benefits are orthogonal to these design choices. Because it works by skipping entire layers during forward execution, it can apply to any underlying CNN accelerator architecture.

The AMC algorithm uses insights from video compression. Specifically, our RFBME algorithm builds on a large body of work on the block-matching motion estimation algorithms that are central to video codecs [55, 20, 19] and their ASIC implementations [56, 57].

In the computer vision literature, the most closely related work is deep feature flow (DFF) [13]. DFF is a neural network design that grafts an optical flow network, FlowNet [24], to a subset of a feature network, ResNet [58], via a spatial warping layer. A recent extension aggregates features across frames to mitigate the effects of motion blur, rare poses, and defocus [14]. The goal is similar to AMC: use motion information to avoid computing a prefix of convolutional layers involved in feature extraction. Our work provides four key benefits over DFF: Firstly, DFF uses a full CNN (FlowNet) to compute per-pixel motion. RFBME performs coarse-grained computation at the receptive-field level. As described in section 4.4.2, FlowNet requires orders of magnitude more computation than RFBME. Second, DFF uses a fixed key frame rate calibrated to the "worst-case scenario" for scene motion. Even with no motion, DFF computes key frames as often as when there is fast motion. AMC's adaptive key frame rate spends less time and energy when frames are more predictable. Thirdly, while DFF stores activations off chip, AMC's compressed activations reduce their size enough for on-chip storage (80-87%). Finally, EVA^2's warp engine skips over zero entries when performing interpolation. This results in computational savings proportional to the sparse compression ratio.

Other vision work has sought to exploit temporal redundancy for efficiency. Clockwork convnets [59] exploit the observation that the values in deeper, more semantic layers change at a slower rate than earlier, noisier layers. The execution strategy uses fixed update rates, however, and does

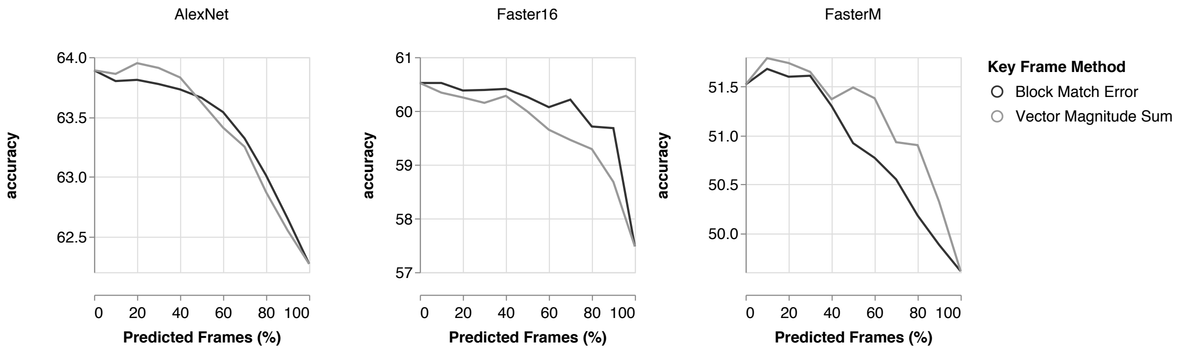


Figure 15: The impact of key frame selection strategy on vision accuracy. The *y*-axes show overall accuracy for mixtures key frames and predicted frames, which use a key frame interval of 198 ms for Faster16 and FasterM and 4891 ms for AlexNet.

not adapt to changes in the input video. *Delta networks* [15, 16] compute the temporal derivative of the input and propagate the change through each layer in a network. Using pixel-level derivatives, however, is a poor match for real video data, where even small movements can cause large changes in pixel values. Section 2 discusses delta networks and their efficiency drawbacks in more detail.

6. CONCLUSION

Generic CNN accelerators leave efficiency on the table when they run real-time computer vision workloads. While this paper exploits temporal redundancy to avoid CNN layer computation, AMC also suggests opportunities for savings in the broader system. Future work can integrate camera sensors that avoid spending energy to capture redundant data [60, 61, 62, 63] and full applications that use vision results can inform the system about which semantic changes are relevant for a specific tasks. Through holistic co-design, approximately incremental vision can enable systems that spend resources in proportion to relevant events in the environment.

7. REFERENCES

[1] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting vision processing closer to the sensor," in *International Symposium on Computer Architecture (ISCA)*, 2015.

[2] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2016.

[3] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *International Symposium on Computer Architecture (ISCA)*, 2017.

[4] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[5] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *International Conference on Learning Representations (ICLR)*, 2016.

[6] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[7] U. A. Acar, *Self-adjusting Computation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005.

[8] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster, "Adapton: Composable, demand-driven incremental computation," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[9] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning approximation for graphics engines," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[10] M. Ringenberg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, "Monitoring and debugging the quality of results in approximate programs.," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[11] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *International Symposium on Computer Architecture (ISCA)*, 2015.

[12] Y.-W. Huang, C.-Y. Chen, C.-H. Tsai, C.-F. Shen, and L.-G. Chen, "Survey on block matching motion estimation algorithms and architectures with new results," *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, vol. 42, pp. 297–320, Mar. 2006.

[13] X. Zhu, Y. Xiong, J. Dai, L. Yuan, and Y. Wei, "Deep feature flow for video recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[14] X. Zhu, Y. Wang, J. Dai, L. Yuan, and Y. Wei, "Flow-guided feature aggregation for video object detection," in *IEEE International Conference on Computer Vision (ICCV)*, 2017.

[15] P. O'Connor and M. Welling, "Sigma delta quantized networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[16] D. Neil, J. H. Lee, T. Delbruck, and S.-C. Liu, "Delta networks for optimized recurrent network computation," in *International Conference on Machine Learning (ICML)*, 2017.

[17] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[18] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?," in *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[19] S. Zhu and K.-K. Ma, "A new diamond search algorithm for fast block matching motion estimation," *IEEE Transactions on Image Processing*, vol. 9, pp. 287–290, Feb. 1997.

[20] R. Li, B. Zeng, and M. L. Liou, "A new three-step search algorithm for block motion estimation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 4, no. 4, pp. 438–442, 1994.

[21] B. S. Reddy and B. N. Chatterji, "An FFT-based technique for translation, rotation, and scale-invariant image registration," *IEEE Transactions on Image Processing*, vol. 5, no. 8, pp. 1266–1271, 1996.

[22] B. D. Lucas and T. Kanade, “An iterative image registration technique with an application to stereo vision,” in *International Joint Conference on Artificial Intelligence (IJCAI)*, 1981.

[23] B. K. Horn and B. G. Schunck, “Determining optical flow,” *Artificial intelligence*, vol. 17, no. 1-3, pp. 185–203, 1981.

[24] P. Fischer, A. Dosovitskiy, E. Ilg, P. Häusser, C. Hazirbas, V. Golkov, P. van der Smagt, D. Cremers, and T. Brox, “FlowNet: Learning optical flow with convolutional networks,” in *IEEE International Conference on Computer Vision (ICCV)*, 2015.

[25] E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy, and T. Brox, “FlowNet 2.0: Evolution of optical flow estimation with deep networks,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[26] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[27] J. Johnson, A. Karpathy, and L. Fei-Fei, “DenseCap: Fully convolutional localization networks for dense captioning,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[28] S. Yeung, O. Russakovsky, G. Mori, and L. Fei-Fei, “End-to-end learning of action detection from frame glimpses in videos,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[29] N. Muralimanohar, A. Shafiee, and V. Srinivas, “CACTI 6.5.” `https://github.com/HewlettPackard/cacti`.

[30] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE Journal of Solid-State Circuits*, vol. 52, pp. 127–138, Jan. 2017.

[31] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” 2014. `http://arxiv.org/abs/1409.1556`.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[33] A. Sampson and M. Buckler, “FODLAM, a first-order deep learning accelerator model.” `https://github.com/cucapra/fodlam`. Commit `cde53fd`.

[34] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards real-time object detection with region proposal networks,” in *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[35] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman, “Return of the devil in the details: Delving deep into convolutional nets,” in *British Machine Vision Conference*, 2014.

[36] E. Real, J. Shlens, S. Mazzocchi, X. Pan, and V. Vanhoucke, “YouTube-BoundingBoxes: A large high-precision human-annotated data set for object detection in video,” 2017.

[37] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[38] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” 2014. `http://arxiv.org/abs/1408.5093`.

[39] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc 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, 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, “In-datacenter performance analysis of a Tensor Processing Unit,” in *International Symposium on Computer Architecture (ISCA)*, 2017.

[40] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” 2017. `https://arxiv.org/abs/1703.09039`.

[41] A. Sodani, “Knights Landing (KNL): 2nd generation Intel Xeon Phi processor,” in *HotChips*, 2015.

[42] D. Foley and J. Danskin, “Ultra-performance Pascal GPU and NVLink interconnect,” *IEEE Micro*, vol. 37, pp. 7–17, Mar. 2017.

[43] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[44] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, “Origami: A convolutional network accelerator,” in *Great Lakes Symposium on VLSI*, 2015.

[45] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, “Minerva: Enabling low-power, highly-accurate deep neural network accelerators,” in *International Symposium on Computer Architecture (ISCA)*, 2016.

[46] M. Alwani, H. Chen, M. Ferdman, and P. Milder, “Fused-layer CNN accelerators,” in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[47] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-based accelerator design for deep convolutional neural networks,” in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015.

[48] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, “Going deeper with embedded FPGA platform for convolutional neural network,” in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[49] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, “Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks,” in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[50] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, “From high-level deep neural models to FPGAs,” in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[51] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, “Scalpel: Customizing DNN pruning to the underlying hardware parallelism,” in *International Symposium on Computer Architecture (ISCA)*, 2017.

[52] C. Zhu, S. Han, H. Mao, and W. J. Dally, “Trained ternary quantization,” in *International Conference on Learning Representations (ICLR)*, 2017.

[53] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks,” in *Advances in Neural Information Processing Systems (NIPS)*, 2016.

[54] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “XNOR-Net: ImageNet classification using binary convolutional neural networks,” in *European Conference on Computer Vision (ECCV)*, 2016.

[55] C. Je and H.-M. Park, “Optimized hierarchical block matching for fast and accurate image registration,” *Signal Processing: Image Communication*, vol. 28, no. 7, pp. 779–791, 2013.

[56] M. Tikekar, C.-T. Huang, C. Juvekar, V. Sze, and A. P. Chandrakasan, “A 249-Mpixel/s HEVC video-decoder chip for 4K ultra-HD applications,” *IEEE Journal of Solid-State Circuits*, vol. 49, no. 1, pp. 61–72, 2014.

[57] M. Tikekar, V. Sze, and A. Chandrakasan, “A fully-integrated energy-efficient H.265/HEVC decoder with eDRAM for wearable devices,” in *2017 Symposium on VLSI Circuits*, pp. C230–C231, June 2017.

[58] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[59] E. Shelhamer, K. Rakelly, J. Hoffman, and T. Darrell, “Clockwork convnets for video semantic segmentation,” in *Video Semantic Segmentation Workshop at European Conference in Computer Vision (ECCV)*, 2016.

[60] M. Buckler, S. Jayasuriya, and A. Sampson, “Reconfiguring the imaging pipeline for computer vision,” in *IEEE International Conference on Computer Vision (ICCV)*, 2017.

[61] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, “RedEye: Analog convnet image sensor architecture for continuous mobile vision,” in *International Symposium on Computer Architecture (ISCA)*, pp. 255–266, IEEE Press, 2016.

[62] H. G. Chen, S. Jayasuriya, J. Yang, J. Stephen, S. Sivaramakrishnan, A. Veeraraghavan, and A. Molnar, “Asp vision: Optically computing the first layer of convolutional neural networks using angle sensitive pixels,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 903–912, 2016.

[63] A. Mazumdar, T. Moreau, S. Kim, M. Cowan, A. Alaghi, L. Ceze, M. Oskin, and V. Sathe, “Exploring computation–communication tradeoffs in camera systems,” in *IEEE International Symposium on Workload Characterization (IISWC)*, 2017.