HyperCam: Low-power Onboard Computer Vision for IoT Cameras

Chae Young Lee[†], Pu (Luke) Yi[†], Maxwell Fite[‡], Tejus Rao[‡],
Sara Achour[†‡], Zerina Kapetanovic[‡]

[†]Department of Computer Science, Stanford University
[‡]Department of Electrical Engineering, Stanford University

ABSTRACT

We present HyperCam, an energy-efficient image classification pipeline that enables computer vision tasks onboard low-power IoT camera systems. HyperCam leverages hyperdimensional computing-based techniques to perform both training and inference efficiently on low-power microcontrollers. We implement a low-power wireless camera platform using off-the-shelf hardware and demonstrate that HyperCam can achieve an accuracy of 93.60%, 84.06%, 92.98%, and 72.79% for MNIST, Fashion-MNIST, Face Detection, and Face Identification tasks, respectively, for 120 × 160 resolution grayscale images. HyperCam performs classification with inference latency of 0.12 s, flash memory usage of about 60 kilobytes, and peak RAM usage of about 20 kilobytes. Among other machine learning classifiers such as SVM, xgBoost, MicroNets, MobileNetV3, and MCUNetV3, HyperCam is the only classifier that achieves competitive accuracy while maintaining competitive memory footprint and inference latency on 4 benchmark tasks.

1 INTRODUCTION

Image sensors are everywhere now, found in smartphones, laptops, gaming consoles, and cars. Coupled with advances in machine learning (ML), object detection and classification can enable practical applications in areas such as healthcare, manufacturing, or transportation. However, ML models, especially deep neural networks (DNNs), require substantial computing power and memory, limiting the adoption of these techniques to Internet-of-Things (IoT). As a result, many IoT cameras offload images to the cloud or gateway, where more compute resources are available [18, 30, 36, 37]. However, this approach introduces overhead in latency, energy consumption, and data privacy. It also causes the system to depend on network communications, which can be costly and unreliable. Recent works have shown that low-power cameras deal with numerous packet losses and, therefore, poor image quality for processing at the base station [18, 30].

A more recent and promising approach is onboard or embedded ML techniques. Some works have shown that deep neural networks (DNNs) can run at the sensor node using accelerated hardware such as MAX78000 and Movidius Myriad 2 [5, 11]. Other works use extensive network architecture search (NAS) to find an optimized neural network model that fits in a microcontroller (MCU) [3, 26]. However, due to the tight resource constraints of embedded devices, ML models perform poorly in vision tasks and are mostly inference-only. For example, in one study demonstrating ML inference underwater, the authors faced challenges in achieving competitive classification accuracy while shrinking the neural network to fit on a MCU [39]. Specifically, low-cost embedded hardware typically lack Floating Point Units (FPUs) and can only execute integer-based models, leading to a drop in the accuracy and precision of ML models [4, 17]. Some advanced hardware do include FPUs but integer-based operations are favored for better energy efficiency. Additionally, ML models require large amounts of training data, which might not be available for relatively smaller-scale IoT tasks.

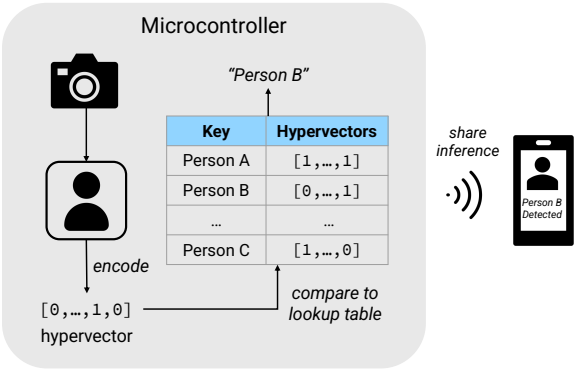

Figure 1: HDC for image classification. HyperCam uses an HD classifier to perform face detection and identification tasks onboard low-power wireless camera platforms.

This paper presents HyperCam, an image processing pipeline that runs computer vision tasks onboard resource-constrained camera systems. As shown in Fig. 1, HyperCam's classifier queries the captured image to its onboard model in real-time and transmits the result to a nearby smartphone via wireless methods. HyperCam leverages hyperdimensional computing (HDC), which is an alternative paradigm in computing built

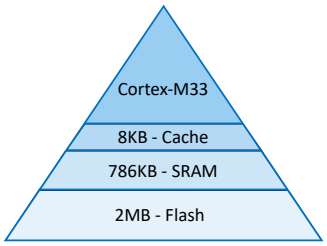

Figure 2: Memory layout of STM32U585AI.

on a set of structured data types and bit operations [22]. Compared to DNNs, HDC is inherently hardware-friendly and energy-efficient [14, 24]. However, most existing HDC works target time-series data because image processing poses several challenges in resource-constrained environments. One is the memory footprint of the HDC model. As shown in Fig. 2, common MCUs have a small and flat memory structure involving a random access memory (RAM) and a non-volatile flash memory. Thus, optimizations are required to reduce the memory footprint of the model in both RAM and flash memory to fit the model onboard. Additionally, optimizing latency is critical, not only to meet real-time requirements but also to minimize the overall power consumption of the system. In image processing, the amount of HD computation load increases proportionally to the image size. For example, a baseline HDC approach can take as long as one minute to classify a grayscale image with 120×160 resolution.

HyperCam solves these challenges using novel and highly efficient HD encoding methods and aggressively optimizing performance in terms of memory and latency. Specifically, it features a lightweight encoder that dynamically maps images into HD space, eliminating the need for pre-stored mappings as other HD classifiers. The encoder also uses a sparse binary bundling based on Bloom Filer and Count Sketch, reducing the number of encoding operations by two orders of magnitude. Integrated into an ARM Cortex M-33 microprocessor, HyperCam is 35.87% more accurate than MobileNet-V3-Small and 43.17% more accurate than MicroNet in the 7-class face identification task. It is also 2-4 times faster and 5-20 times more lightweight than these baseline DNNs. The following are key contributions made in this work.

- We introduce HyperCam, a novel HD image classifier that deploys highly efficient novel data encodings to perform inference on the sensor node. HyperCam is far more accurate, lightweight, and fast than previous HDC methods and DNN baselines.
- We develop a prototype of a low-power wireless camera platform to evaluate HyperCam.
- We show that HyperCam can perform binary and multiclass classifications in real time using captured image frames. HyperCam achieves an accuracy of 92.60% and 60.91% for face detection and identification, respectively, using only about one hundred kilobytes of memory and achieving a latency of 0.1-0.3 seconds.
- We open source the HyperCam code to help promote reproducibility and advance onboard computing methods.

2 HYPERDIMENSIONAL COMPUTING BACKGROUND

Hyperdimensional Computing (HDC), or Vector Symbolic Architectures (VSA), is a brain-inspired computing paradigm that represents information in a high-dimensional space. Within this framework, data is encoded as *hypervectors* — vectors typically consisting of thousands of dimensions. Randomly generated hypervectors called the *basis hypervectors* represent discrete data units such as symbols and numbers. Applying HDC operators such as binding, bundling, and permutation on these basis hypervectors constructs hypervector representations of more complex data structures (e.g., sequences, trees, and images). Information can be retrieved from hypervectors by computing the distances between hypervectors. HDC models vary widely in terms of their choices of hypervector representations, operators, and distance metrics. HyperCam uses the Binary Spatter Code (BSC) approach [20], where each element of a hypervector is binary. Operations done on binary hypervectors are simple and energy-efficient and, thus, the best choice for resource-constrained hardware. In the following sections, along with Fig. 3, BSC-HDC operations are explained in more detail.

2.1 Binary Spatter Code

2.1.1 Basis vector generation. In BSC, each unique symbol is represented as a binary hypervector called the basis hypervector. Each element of the vector is a bit, randomly generated with a $p = 0.5$ Bernoulli. The hyperdimensionality of these vectors ensures that randomly generated vectors are nearly orthogonal to each other. In other words, any two basis hypervectors are far apart, usually with a Hamming distance of about 0.5. These basis hypervectors, also called *codes*, are stored in a dictionary data type called the *codebook*.

2.1.2 Binding. The binding operator ($\odot$) combines basis hypervectors and creates a hypervector dissimilar to the input. In BSC, binding is implemented as an exclusive OR (XOR). Binding is used to construct larger data structures such as composite symbols, key-value pairs, and positional encoding from basis hypervectors. For example, binding the two hypervectors that represent the words `cold` and `water` results in a single hypervector for `cold water`. Similarly, binding the hypervectors for the key and the value creates a hypervector for the key-value pair.

2.1.3 Bundling. The bundling operator ($\oplus$) aggregates multiple hypervectors and outputs a hypervector similar to the

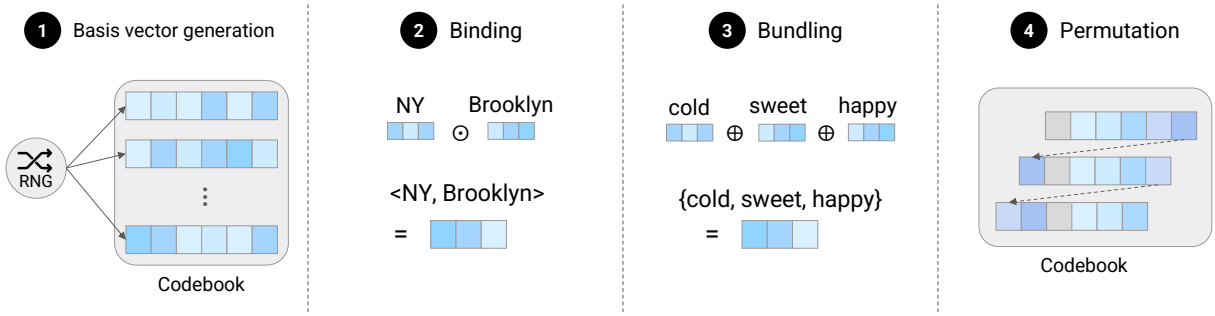

Figure 3: Key operations of BSC. (1) Basis vectors are generated for every letter. **(2)** Binding of data creates a record. **(3)** Bundling of words creates a set. **(4)** Permutation is applied to create hypervectors on-the-fly.

input. In BSC, bundling is executed through an element-wise majority vote. Given two or more input hypervectors, the number of zeros and ones are counted at each index, and the output hypervector chooses the majority value at that index. Bundling is used to create sets of symbols or data instances. For example, an image hypervector is created by bundling the hypervectors of its pixels. Similarly, a hypervector for a database record is created by bundling the hypervectors of its key-value pairs.

2.1.4 Permutation. The permutation operator (ρ) is implemented as a circular shift, which creates a dissimilar hypervector far apart from the input. Because of this characteristic, permutation is used to create new basis hypervectors as an alternative to random generation. Additionally, permutation is used to encode the position data of sequences. For example, a bigram can be encoded by binding the permuted hypervectors of the characters. That is, binding occurs between the hypervector of the first character and the permuted hypervector of the second character. Similarly, the dimension of an image array can be represented using permutation. For 2-dimensional images, binding occurs between the hypervector of the row index and the permuted hypervector of the column index.

2.1.5 Distance metric. While binding, bundling, and permutation operators encode raw data into hypervectors, distance metrics are used to retrieve information from the hyperdimensional space. The lower the distance between two hypervectors, the more similar they are. In BSC, distance measurement is implemented using the Hamming distance, which counts the number of differing bits and normalizes the count by the length of the hypervector. The distance metric is often used to identify the class of the query hypervector. Other times, it is used to decode the hypervector to its raw data form (e.g., sequences, images). For example, the identity of the key in a hypervector for a key-value pair can be determined by computing the distance between the hypervector and all possible key hypervectors.

3 HYPERCAM DESIGN

Using BSC-HDC operations, HyperCam processes computer vision tasks at the endpoint device (e.g., wireless IoT camera). When the camera captures an image, that image is converted into a hypervector, and the classifier determines its class based on the Hamming distances. In addition, HyperCam can send the classification result to an IoT gateway via Bluetooth, allowing remote monitoring and interaction. This approach of transmitting the classification data, as opposed to entire raw images, significantly reduces communication overhead and mitigates the risk of transmission errors. The architecture of the HyperCam classifier, as shown in Fig.4, has three major components: an image encoder, a training algorithm, and an inference algorithm.

Image Encoder. Both inference and training require that images be first encoded as hypervectors. This translation is done by performing an HD computation over basis hypervectors that capture pixel position and value information. The MCU stores codebooks that contain these basis hypervectors. A critical challenge in applying HD classifiers to image classification tasks is managing the performance and memory overhead associated with encoding image data as hypervectors. HyperCam deploys a novel image encoding algorithm (Section 4) that exploits the structure of HD computations to drastically reduce the memory usage and latency of the encoding procedure. This encoding algorithm uses a novel sparse bundling algorithm (Section 4.3) to accelerate the bundling of sets of elements.

Training. Training the HyperCam's HD classifier occurs offline on a commodity computer. In this phase, training data are first encoded to hypervectors through the image encoder. Then, these encoded hypervectors are grouped based on their class labels. The class hypervector is construed by bundling together the hypervectors of all data instances that belong to that class. The table of class hypervectors is called the *item memory*. While HD classifiers are typically trained using a one-shot algorithm, HyperCam uses the OnlineHD

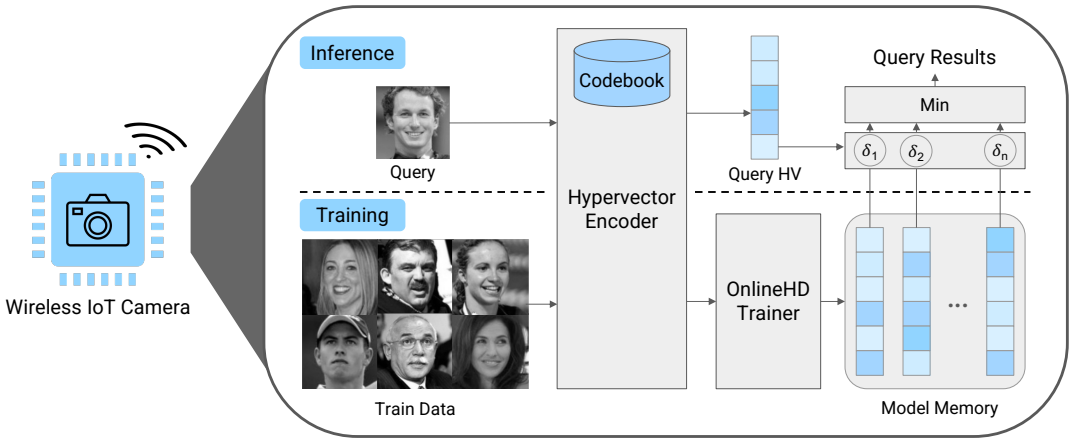

Figure 4: HyperCam overview. The HyperCam classifier runs onboard a low-power wireless camera platform and has three key components: image encoder, training algorithm, and inference algorithm.

adaptive training algorithm [12]. OnlineHD provides an effective few-pass learning approach where classifier hypervectors are refined based on the misclassifications observed on each training iteration. OnlineHD targets MAP-HDC, which works with real-valued vectors. HyperCam works with a modified version of OnlineHD that works with binary hypervectors. The adapted algorithm binarizes the real-valued classifier vectors after each training iteration and uses the binarized item memory to find misclassifications and update the real-valued model.

Inference. During execution, the MCU encodes each input frame to a hypervector. This hypervector, called the query hypervector, is compared to all the class hypervectors in the item memory using the Hamming distances. The class with the smallest distance to this query is the predicted category of the input. This inference computation is highly computationally efficient as it involves only simple Hamming distance calculation.

4 HYPERCAM IMAGE ENCODING

HyperCam's image encoding uses both HD expression optimizations and a novel sparse bundling operator to dramatically reduce the encoding overheads. Section 4.1 presents the naïve image encoding HyperCam's encoding is based on, Section 4.2 presents the rewrites applied to reduce memory and computation requirements, and Section 4.3 presents the novel sparse bundling method HyperCam uses to expedite image encoding.

Table 1 presents the computation and memory requirements of the unoptimized, naïve encoding compared to the optimized encoding employed by HyperCam. The HyperCam encoding is obtained by applying four HD expression rewrites (*Rewrite 1-4*) that progressively reduce the space and computational requirements of the encoder. HyperCam employs a novel sparse bundling algorithm that approximates HDC bundling while requiring 0.2% of the operations. With these algorithmic encoding optimizations, HyperCam's encoding algorithm uses 52% less codebook memory, 50% less binding operations, and 98% less bundling operations. The 19200 sparse bundling operations in the final encoding are computationally equivalent to 38 normal HD bundling operations.

4.1 Naïve HD Image Encoding

This section describes a naïve pixel-based HD image encoding algorithm for grayscale images. HyperCam works with a heavily optimized encoding derived from this naïve encoding. In the following encoding formulations, n, w, and h refer to hypervector size, image width, and height. HyperCam supports encoding of 8-bit grayscale images, and each pixel value $Img[i, j]$ is represented as a value from 0 to 255.

Pixel Position Codebook. The naïve image encoding uses pixel row, column, and value codebooks. The pixel row codebook $R(i)$ maps the pixel rows $i \in 1 \cdots h$ to hypervectors, while the pixel column codebook $C(j)$ maps the pixel columns $j \in 1 \cdots w$ to hypervectors. These codebooks map each row and column index to a randomly generated binary hypervector.

Pixel Value Codebook. The pixel value codebook $V(x)$ maps an 8-bit grayscale value $x \in 0 \cdots 255$ to a hypervector. The hypervectors in the pixel value codebook are generated using a variant of level-based encoding [32]. The level-based encoding method ensures that hypervectors representing similar grayscale values have small Hamming distances. In this level-based codebook, $V(0)$ is instantiated to a zero vector of length n representing a black pixel value. The basis

	Naive		Rewrite 1		Rewrite 2		Rewrite 3		HyperCam	
	sym	act	sym	act	sym	act	sym	act	sym	act
Codebook	$w+h+256$	536	$\lceil \frac{w}{n} \rceil + \lceil \frac{h}{n} \rceil + 256$	258	$\left\lceil \frac{wh}{n} \right\rceil + 256$	258	$\left\lceil \frac{wh}{n} \right\rceil + 256$	258	$\left\lceil \frac{wh}{n} \right\rceil + 256$	258
Bind	$2wh$	38400	$2wh$	38400	wh	19200	wh	19200	wh	19200
Bundle	wh	19200	wh	19200	wh	19200	$wh+256$	19456	256	256
SparseBundle	0	0	0	0	0	0	0	0	wh	19200

Table 1: Comparison of different encoding methods. The table provides a row-wise comparison of the number of hypervectors in the codebook and the number of binding, bundling, and sparse bundling operations for each encoding method in both symbolic expressions (sym) and actual numbers (act).

hypervectors for values $1, \cdots, 255$ are constructed by sequentially setting random selections of zero-valued bits to one.

Pixel Encoding. Given a 1D pixel array Img, the naïve image encoding converts the grayscale value $Img[i \cdot w + j] = x$ of a pixel at location i, j to a pixel hypervector by binding the three separate hypervectors representing the pixel's row and column index and the pixel's grayscale value into one hypervector:

$$hv_{pix,i,j} = R(i) \odot C(j) \odot V(Img[i \cdot w + j])$$

Image Encoding. An image hypervector hv_{img} is then constructed from the pixel hypervectors by bundling the pixel hypervectors together:

$$hv_{img} = \sum_{i=1}^{h} \sum_{j=1}^{w} R(i) \odot C(j) \odot V(Img[i \cdot w + j])$$

With the above encoding, image hypervectors containing similar pixels will have small hamming distances.

Space and Time Complexity. The above encoding method needs to store $w + h + 256$ codebook hypervectors, each of which is $n = 10000$ bits. The naïve encoding method requires wh pixel bundling operations and $2wh$ binding operations per image, each of which is a n-bit hypervector operation. For the 120×160 grayscale images used in this implementation, naïve encoding would require 536 codebook hypervectors totaling 670 kilobytes of memory, 38400 binding operations, and 19200 bundling operations. Therefore, even for small images, this encoding algorithm scales poorly.

4.2 HyperCam HD Image Encoding

Based on the naïve encoding method presented in Section 4.1, the properties of HD computations are exploited to rewrite the image encoding and optimize computation and memory usage. Sections 4.2.1-4.2.4 present the HD expression rewrites applied to reduce the image encoder's memory footprint and computational requirements. The rewrites presented in 4.2.1-4.2.2 preserve the computational properties of the HD encoding and therefore do not affect classification accuracy. The factoring and sparse bundling rewrites in 4.2.3-4.2.4 are semantics-breaking and change the computational properties of the HD encoding, therefore affecting classification accuracy. HyperCam's sparse bundling optimization uses a novel lossy filter-based sparse bundling operator, which is presented in Section 4.3.

4.2.1 Rewrite 1: Permutation-based Codebooks. First, HyperCam uses the permutation operator described in Section 2.1.4 to encode row and column index information, eliminating $h-1$ and $w-1$ entries from row and column hypervector codebooks, R and C, respectively:

$$hv_{img} = \sum_{i=1}^{h} \sum_{j=1}^{w} p^i[R(0)] \odot p^j[C(0)] \odot V(Img[i \cdot w + j])$$

Since the hypervectors in the pixel position codebook are generated independently and randomly, permuting one code effectively produces another independent code. In other words, any code and its permuted code have a high expected distance similar to that of two independent codes. Therefore, with the permutation operator, fewer hypervectors can be used in place of a random codebook. The row and column index determines how many times the zero-index row and column hypervectors $R(0)$ and $C(0)$ are permuted. This optimization reduces the number of codebook hypervectors from $w + h + 256$ to $\lceil \frac{w}{n} \rceil + \lceil \frac{h}{n} \rceil + 256$ hypervectors, thus reducing codebook memory usage from 670 KB to 322 KB.

4.2.2 Rewrite 2: Row and Column Index Coalescing. In naïve encoding, the row and column hypervectors are bound together to produce a positional hypervector that encodes the pixel coordinate. This binding operation between row and column hypervectors can be eliminated and replaced with a new random hypervector codebook X that captures 1D pixel position:

$$hv_{img} = \sum_{i=1}^{h} \sum_{j=1}^{w} X(i \cdot w + j) \odot V(Img[i \cdot w + j])$$

This rewrite can be applied because the binding operator produces a hypervector that is dissimilar to its input hypervectors, and the input hypervectors are independent of one another. Therefore, the 2D (i, j) image coordinates can be encoded with a single 1D pixel position $k = i \cdot w + j$ while still preserving the behavior of the HD image encoding. The permutation rewrite can then be applied to reduce the codebook from $w + h$ entries to $\lceil \frac{wh}{n} \rceil$ entries:

$$hv_{img} = \sum_{i=1}^{h} \sum_{j=1}^{w} p^{i \cdot w + j}[X(0)] \odot V(Img[i \cdot w + j])$$

This optimization reduces the number of binding operations per pixel from 2 to 1. With the permutation rewrite applied, the X codebook requires $\lceil \frac{wh}{n} \rceil$ hypervectors.

4.2.3 Rewrite 3: Value Hypervector Factoring. Critically, the number of bundling operations must be reduced to encode the image efficiently. The sparse bundling operator efficiently approximates bundling operations over permutations of a single hypervector. To use sparse bundling, the value hypervector binding operations are factored from the bundling operation:

$$hv_{img} = \sum_{z=0}^{255} V(z) \odot \left[\sum_{i,j \in Pix(z)} p^{i \cdot w + j}[X(0)] \right]$$

$Pix(z)$ returns all pixel positions i, j where each pixel $Img[i \cdot w + j]$ has the value z. Note that the HD $\oplus$ operation is not associative since some information is lost during quantization step of bundling. Thus, this rewrite changes the distance properties of the encoded hypervectors. Specifically, this rewrite loses information about the relative prevalence of different pixel values in the image. For example, if an image contains one gray pixel and many white pixels, the white and gray pixels would be equally important in this factored encoding. This information is re-introduced into the encoding using a weighted bundling: more prevalent pixel values are bundled multiple times.

$$hv_{img} = \sum_{z=0}^{255} |Pix(z)| \cdot V(z) \odot \left[\sum_{i,j \in Pix(z)} p^{i \cdot w + j}[X(0)] \right]$$

Here, values that occur more frequently in the image are heavily weighted in the encoding, recouping some information lost in the factored operation. This weighted bundling operation is equivalent to an HD bundling operation, where each hypervector is bundled multiple times:

$$\sum_{z=0}^{255} |Pix(z)| \cdot [V(z) \cdots] = \sum_{z=0}^{255} \sum_{k}^{|Pix(z)|} [V(z) \cdots]$$

Therefore, the weighted bundling operation can easily be fused with a normal bundling operation by scaling the binary hypervector during the sum-threshold computation.

4.2.4 Rewrite 4: Sparse Bundling. After applying the factoring rewrite, each pixel bundling sub-computation (blue text) can then be efficiently approximated using a novel sparse bundling operator introduced in Section 4.3:

$$hv_{img} = \sum_{z=0}^{255} |Pix(z)| \cdot V(z) \odot \left[\sum_{i,j \in Pix(z)} p^{i \cdot w + j}[X(0)] \right]$$

The sparse bundling operation approximates the standard HD bundling and replaces bundling operations over permuted hypervectors. It is designed to preserve the property of bundling that similar sets of pixels are embedded into hypervectors that are close to each other. It also processes a set of elements (in this case, pixel positions) and returns a hypervector that approximates the distance properties of the standard bundled set of elements:

$$hv_{img} = \sum_{z=0}^{255} |Pix(z)| \cdot V(z) \odot SparseBundle(Pix(z)) \quad (1)$$

The sparse bundling operator works with a density parameter d, where $d \ll n$, and performs $O(d)$ operations to bundle two hypervectors. Using a sparse bundling operation reduces the number of operations required to bundle each vector from $O(n)$ to $O(d)$. HyperCam uses $d = 100$, thus reducing the number of operations per bundling operator from 10000 to 20 operations. Once sparse bundling is applied to construct each pixel set hypervector, HyperCam applies 256 weighted HD bundling operations to construct the final hypervector representation of the image.

4.3 Sparse Bundling

HyperCam deploys a novel sparse bundling algorithm that uses Bloom Filter [6] and Count Sketch [8] to approximately bundle large numbers of hypervectors together at low latency. Bloom Filters and Count Sketches are probabilistic data structures adept at representing sets of elements. Both data structures work with numeric vectors and are updated by randomly sampling and updating bits. They can be viewed as a sub-class of HDC/VSA as they also compute in superposition [9, 21, 23].

Given a set of integers $s \in S$, the sparse bundling algorithm returns a binary hypervector that approximates bundling together the basis hypervectors that represent each element:

$$SparseBundle(S) \approx \sum_{s \in S} p^s[hv]$$

Algorithm 1 Sparse Bundling Algorithm

```
1:  bool bloom = false; //use a bloom filter or count-sketch
2:  uint n = 10000; // hypervector size
3:  uint d = 20; // density - the number of hashes per bundle
4:  // random set of size d from values {0, 1, ··· , n − 1}
5:  uint8[d] indices ← rand(0,n,size=d,replace=False);
6:  // random vector with values {−1, 1}
7:  int8[d] CS ← rand([-1,1],size=d);
8:  function SparseBundleElem(hv,s)
9:    for j in 0...d − 1 do
10:     k = (indices[j]+s) % n
11:     if bloom then
12:       hv[k] = 1
13:     else
14:       hv[k] = hv[k] + CS[j]
15: function NewSparseHV
16:   int8[n] hv = zeroes(n);
17:   return hv;
18: function FinalizeHV(hv)
19:   if ¬ bloom then
20:     for i in 0..n do
21:       hv[i] = 1 ? hv[i] >= 0 : 0
22: procedure SparseBundle(S)
23:   hv = NewSparseHV()
24:   for s ∈ S do
25:     SparseBundleElem(hv,s)
26:   FinalizeHV(hv);
27:   return hv;
```

The sparse bundling operation approximates an HD bundling operation over permutations (p^s) of some hypervector hv. The algorithm is parametrized with a hypervector size n and density parameter d and offers both Bloom Filter and Count Sketch backends. The Bloom Filter backend is more computationally efficient but less accurate than the Count Sketch backend. Each sparse bundling operation requires d operations, significantly reducing the number of operations per bundling task when $d << n$.

4.3.1 Algorithm Description. This section describes Alg. 1. Given a set of integer values S to bundle, the *SparseBundle* operator instantiates a new sum hypervector (Line 23), uses the sparse bundling operator to add each value to the sum hypervector (Lines 24-25), and then finalizes the sum hypervector (Line 26) to obtain a binary hypervector that approximates the bundled result. The *SparseBundleElem* routine updates the sum hypervector to bundle an integer element.

Instantiation and Finalization. The sum hypervector is instantiated to an n-dimensional signed integer vector comprised of zeroes. On finalization, each element is binarized by thresholding the value with zero to produce an n-dimensional binary vector. Finalization is only required for sum hypervectors in the Count Sketch backend; the Bloom Filter backend directly produces binary vectors.

Bundling[Lines 8-14] The algorithm updates the sum hypervector hv to include the integer s by computing d random indices from the integer value and then updating the values in these indices. For the Bloom Filter backend, each update sets the hypervector value to one. For the Count Sketch backend, the bundle hypervector value is randomly incremented or decremented. HyperCam precomputes the random indices, along with the random increment and decrement operations, and stores the values in the Count Sketch (CS) array.

5 HYPERCAM IMPLEMENTATION

In this section, the implementation of HyperCam is described. Section 5.1 presents the implementation of the image encoding algorithm and further engineering efforts to port the model onboard. Section 5.2 describes the collection of the image dataset used to evaluate HyperCam. Lastly, Section 5.3 describes HyperCam's low-power hardware platform.

5.1 Image Encoding Algorithm

Algorithm 2 Image Encoding with Sparse Bundling

```
function encodeImage(Img: image)
  int[256][n] hvs;
  uint[256] cnts;
  uint8[n] imgHV;
  for v in 0..256 do
    sumHVs[v] = NewSparseHV()
    cnts[v] = 0
  for k in 0..w · h do
    v = Img[k]
    SparseBundleElem(sumHVs[v], k)
    cnts[v] += 1
  for v in 0..255 do
    for i in 0 ··· n − 1 do
      imgHV[i] += cnts[v] · (sumHVs[v][i] xor getValueCB(v,i))
  for i in 0 ··· n − 1 do
    imgHV[i] = 1 ? imgHV[i] > |wh|/2 : 0;
  return imgHV;
```

Alg. 2 presents the algorithm for computing the optimized image encoding presented in Equation 1.

First, a single pass is taken over the input image, during which the position hypervectors are bundled using either

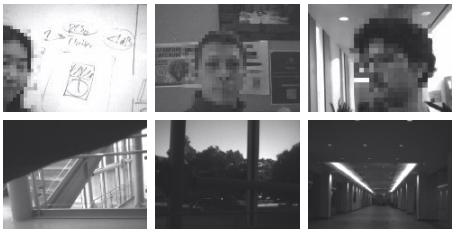

Figure 5: Sample images in collected dataset.

a Count-Sketch-based or Bloom-Filter-based bundling operation. The binary hypervectors produced by the sparse bundling operation are then bound with the value hypervectors and bundled to form the final image hypervector, as described in Equation 1. Each bundled vector is bound with the corresponding value hypervector from the codebook, resulting in 256 hypervectors. These hypervectors are then bundled together, using weights equivalent to the number of pixels in each bin. Moreover, since each hypervector contains only d non-zero elements, the vector summation in the binning process computes d integer elements instead of n.

To further reduce image encoding time and eliminate value codebook, value hypervectors are generated on-the-fly instead of pre-storing them. As described in Section 4.1, the value codebook uses level-based encoding, where random selections of bits in $V(0)$ are flipped. For a value v, $V(v)$ is generated by flipping $v \cdot \lfloor n/256 \rfloor$ bits. The order in which bits are flipped must be the same at every generation for encoding consistency. Thus, this ordering of bit flips (which are indices of the n-length array) is stored in the microcontroller. Additionally, the generation of value hypervectors does not create overhead in computation because it integrates into the binding operation, which already iterates over the vector.

Complexity. These implementation-level optimizations reduce the codebook size from 256 + 2 to 2 hypervectors. The above algorithm requires 256 bundling operations, 256 binding operations, and 19200 fast bundling operations. Each fast bundling operation uses approximately 500x fewer operations than standard HDC bundling.

5.2 Data Collection

There are existing datasets for both face detection and identification, such as the DigiFace-1M or the VGGFace2 dataset [2, 7]. However, these open-source images are either synthetic or captured by cameras and settings far different from our deployment environment. Thus, a dataset was created for the task of interest. Specifically, we took over 1000 images of size 160x120 using the Himax HM01B0 camera mounted on the Arducam HM01B0 Monochrome SPI Module. An assortment of backgrounds and people was imaged to diversify the input dataset. Images of people were taken such that their faces were captured at different angles and positions within the image frame. Approximately 500 images per person were collected from different background scenes such as a hallway, office space, whiteboard, etc. Objects and backgrounds not involving people were also collected as negative samples for face detection. There are a total of 4,215 images across 7 person classes and 1 non-person class. Fig. 5 shows an example of six images that were collected as part of the data set.

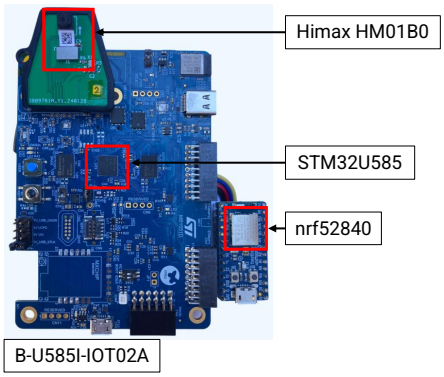

Figure 6: Low-power wireless camera platform.

5.3 Hardware

To evaluate the performance of HyperCam on resource-constrained hardware, a low-power camera hardware platform was designed. An evaluation board for the STM32UF855AI microcontroller (MCU) was used as the central computing device [34, 35]. The MCU has an Arm Cortex-M33 processor, 2MB of flash memory, and 736KB of SRAM. The evaluation board contains several sensors, extra memory, and peripheral interfaces that are redundant for the evaluation of HyperCam. Thus, all non-critical components operating on the same power supply rails as the MCU were removed from the board to reduce power consumption. The Himax HM01B0 image sensor in QQVGA mode is used to capture 160x120 resolution grayscale images [13]. A custom printed circuit board (PCB) is implemented to interface the MCU with the image sensor, which connects the 2.8V supply from the evaluation board to the camera. Moreover, it connects I2C and 8-bit parallel QQVGA communications between the MCU and image sensor. A 24MHz crystal oscillator drives the image sensor's internal clock. Lastly, the MCU's Digital Camera Interface (DCMI) and Direct Memory Access (DMA) peripherals are used to transfer image data from the camera into the MCU's memory.

A nRF52840 BLE module is integrated into the camera hardware and is used to wirelessly transmit data packets to a nearby base station [33]. A 3.7V 4400mAh Lithium Ion battery powers the entire hardware platform. Two linear regulators on the evaluation board provide 3.3V and 2.8V to the MCU and camera, respectively. A 3.3V linear regulator also supplies the BLE module. Some power losses were incurred

Table 2: Comparison of Image Classifiers. HyperCam is compared with different image classifiers in terms of accuracy (%), flash memory usage (KB), peak RAM memory usage (KB), and latency (s) on 4 benchmark tasks.

Type		MNIST				Fashion MNIST				Face Detection				Face Identification			
		Acc	Flash	RAM	Latency	Acc	Flash	RAM	Latency	Acc	Flash	RAM	Latency	Acc	Flash	RAM	Latency
HDC	Vanilla HDC	80.03	-	-	-	69.39	-	-	-	72.54	-	-	-	40.60	-	-	-
	OnlineHD	91.34	-	-	-	81.83	-	-	-	84.62	-	-	-	84.62	-	-	-
	Rewrite 2	94.60	365.02	22.09	0.21	84.99	365.02	22.09	0.21	94.09	356.5	22.09	11.56	78.63	362.60	22.09	11.56
	HyperCam*	93.60	63.00	22.25	0.26	84.06	63.00	22.25	0.26	92.98	53.83	22.25	0.27	72.79	59.52	22.25	0.27
	HyperCam**	90.36	52.62	22.25	0.08	83.10	52.62	22.25	0.08	92.73	42.91	22.25	0.12	61.40	49.00	22.25	0.12
Lightweight ML	SVM	78.24	-	-	-	72.06	-	-	-	86.45	-	-	-	27.07	-	-	-
	xgBoost	76.86	365.55	77.09	0.01	71.76	352.76	77.09	0.01	94.46	134.92	77.09	0.01	38.88	193.24	77.09	0.01
Neural Networks	MicroNets	97.82	582.16	302.87	1.05	86.84	582.16	302.87	1.05	92.86	581.12	502.87	6.64	51.71	581.76	502.87	6.64
	MobileNet V3	98.69	1640.00	302.87	3.29	86.48	1640.00	302.87	3.29	88.18	1640.00	502.87	18.53	51.28	1640.00	502.87	18.55
	MCUNet V3*	99.34	1190.00	302.91	6.70	93.3	1190.00	302.91	6.70	99.88	1190.00	302.91	6.70	99.15	1190.00	302.91	6.70
	MCUNet V3**	98.97	1340.00	502.91	46.71	94.20	1340.00	302.91	46.71	99.88	1340.00	502.91	46.71	99.01	1340.00	502.91	46.71

in these linear regulators during the active state, which can be reduced by custom power management design.

The camera platform is designed so that components are set to standby in their minimum-sleep modes and are activated by an event trigger (e.g., motion detection or manual button press). When triggered, the MCU wakes up the camera module to capture and store an image, performs image classification, and transmits the classification outcome to a smartphone application. After completing each task, the camera platform returns to sleep mode. Figure 6 shows the prototype implementation of the hardware platform.

6 EVALUATION

HyperCam is compared with baseline machine learning classifiers in an identical embedded hardware environment.

6.1 Experimental Setup

6.1.1 Classifier Tasks. HyperCam is evaluated on four image classification tasks: MNIST, Fashion MNIST, Face Detection, and Face Identification. MNIST and Fashion MNIST are widely used benchmark datasets for evaluating machine learning classifiers, each containing 60,000 grayscale images of size 28x28 [25, 38]. In contrast, the Face Detection and Identification tasks utilize the collected dataset described in Section 5.2, which consists of 8 classes: 1 non-person class (objects and places) and 7 person classes. The Face Detection task is a binary classification distinguishing between the non-person class and the person class, while the Face Identification task involves a 7-class classification among the 7 person classes. To ensure fair training and testing, all class sizes were balanced, with each class having a similar number of images. Here, the benchmark datasets (MNIST and Fashion MNIST) are used to compare general approaches, whereas the custom tasks demonstrate an IoT deployment scenario where models handle lower quality and smaller datasets.

6.1.2 Classifiers. Several machine learning algorithms are selected as baseline to compare against HyperCam. Both HyperCam and the baseline models are trained offline on a standard laptop, where their test accuracies are assessed. The trained models are then exported as C header files and loaded onto STM32U585AI for performance evaluation. All models use integer representations to fit the hardware and ensure compatibility with other MCU families. Except for the HDC models, which are inherently integer models, all other ML models were trained using floating-point numbers and then quantized post-training to integer values.

HDC. Three baseline HDC models are assessed against two versions of HyperCam. For baseline, VanillaHDC is the most basic form of a HD classifier explained in Section 4.1. `OnlineHD` uses the OnlineHD [12] training method on top of `VanillaHDC`. `Rewrite2` uses the encoding method described in Section 4.2.2 and uses the OnlineHD training method. On the other hand, HyperCam is assessed in two versions: HyperCam* that uses the `count-sketch` backend and HyperCam** that uses the `bloom-filter` backend. All HyperCam models use the OnlineHD training method as well.

Lightweight ML. `SVM` and `XGBoost` are chosen to represent lightweight ML models. They are trained using Python's sklearn and xgboost libraries and are ported to a C header file using the micromlgen library.

Neural Networks. `MicroNets`, `MobileNetV3`, and two sizes of `MCUNetV3` are chosen for this category. MCUNet V3* (mcunet-in1) is the smallest and the MCUNet V3** (mcunet-in3) is the largest one that fits the MCU. They are trained from

pre-trained weights and are quantized to 8-bit integer numbers after training. Once converted to C header files, the `TensorFlow Lite Micro` and the `CMSIS-NN` libraries are used to run them on the ARM Cortex M-33 environment.

6.1.3 Evaluation Metrics. These metrics were used:

Accuracy. Data is split in an 8:2 ratio between the training and testing datasets. The model is trained with the training dataset and accuracy is measured with the test dataset.

Flash Memory. The flash memory footprint of the model is measured in kilobytes. For ML models, this includes the model weights, parameters, and the library code required for execution. For HDC models, this includes the model's codebook and the item memory required for encoding and prediction.

RAM. The peak RAM footprint of the model is measured in kilobytes. For ML models, this includes the model activations, input, and output tensors, and library code. For HDC models, this includes hypervectors allocated for encoding. When encoding is done, HDC uses only one hypervector to represent a data instance for inference.

Latency. The latency of the classifier is the time it takes to process one frame of image. This involves the time it takes to encode an image and predict its class using the item memory. All latency is measured on STM32U585AI and is in seconds.

6.2 Classifier Evaluation

Table 2 compares HyperCam's HD classifier to the baseline in terms of accuracy, flash memory size, peak RAM size, and latency during one pass of inference. The `VanillaHDC` and `OnlineHD` classifiers, while demonstrating reasonable accuracy (80.03% and 91.34% on MNIST, respectively), are unsuitable for deployment on resource-constrained devices due to their large flash memory requirements. HyperCam, however, uses the `Rewrite2` encoding method to significantly reduce the flash memory consumption to 365.02 KB for the largest task while maintaining a competitive accuracy of 94.60% on MNIST and 84.06% on Fashion MNIST.

The final version of HyperCam further improves this by achieving the lowest flash memory footprint of all ML classifiers: 63.00 KB (HyperCam*) and 52.62 KB (HyperCam**) for the largest task. For a more competitive memory footprint and latency, HyperCam sacrifices accuracy from `Rewrite2` but only with a small margin (1.00% reduction in MNIST and 0.93% in Fashion MNIST). HyperCam** also achieves the lowest latency across all HDC and neural network models, with 0.08 seconds on MNIST and 0.12 seconds on both Face Detection and Identification tasks.

When compared to lightweight machine learning models like `SVM` and `xgBoost`, HyperCam demonstrates superior performance in both accuracy and memory efficiency. For example, both versions of HyperCam achieve higher accuracy than `SVM` across all classification tasks, while `xgBoost` only outperforms HyperCam in the Face Detection task by a small margin of 1.48%. In the Face Identification task, `SVM` and `xgBoost` experience a significant drop in accuracy (27.07% and 38.88%, respectively). By contrast, all HD classifiers, including HyperCam, as well as `MCUNetV3`, exhibit a more graceful decline in performance, maintaining much higher accuracy levels (72.79% for HyperCam* and up to 99.15% for `MCUNetV3`). Additionally, in terms of memory consumption, both `SVM` and `xgBoost` require significantly more memory than HyperCam. Even after being quantized to integer values, `SVM` was still too large to fit within the MCU's flash memory.

Neural network models, particularly those using 8-bit integer quantization, such as `MicroNets` and `MobileNetV3`, offer the highest accuracy levels (e.g., 98.69% on MNIST for `MobileNetV3`) but at the cost of substantially higher memory usage and latency. For example, in terms of flash memory consumption, `MicroNets` requires over 500 KB of flash memory while `MobileNetV3` and `MCUNetV3` all use more than 1 MB of flash memory. On the other hand, HyperCam's most memory-efficient version (`count-sketch`) requires only 63 KB of flash memory and 22.25 KB of RAM. `MCUNetV3` exhibits the highest accuracy among all models, with near-perfect performance (e.g., 99.34% on MNIST and 99.88% on Face Detection). However, the trade-off comes in the form of substantially higher latency. The smallest `MCUNetV3` model has a latency of 6.7 seconds, while the larger version takes up to 46.71 seconds. In contrast, HyperCam maintains latencies under 0.3 seconds across all datasets.

Memory Analysis. A breakdown of the flash memory is shown in Fig. 7. In both MobileDNN and EdgeDNN, memory is consumed by the TF Micro library and the model's graph data on top of the model weights and parameters. Using an external library indicates that the hardware choices are restricted to those the library supports. On the other hand,

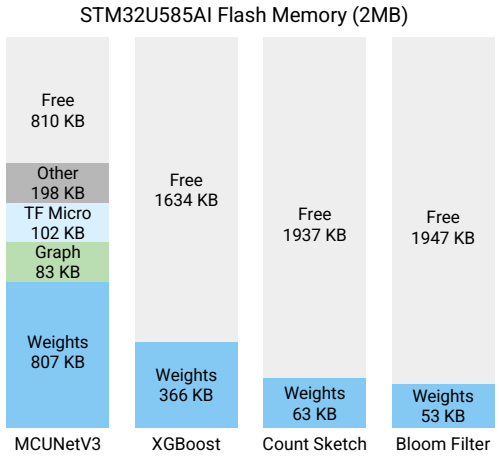

Figure 7: Breakdown of flash memory.

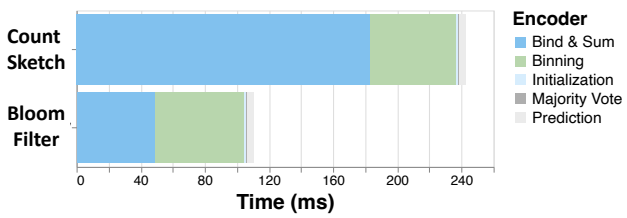

Figure 8: Latency Profiling.

Component	Active Current	Sleep Current	Voltage
STM32U585I	10.9 mA	74.9 μA	3.3V
Himax HM01B0	2.5 mA	1.3 mA	2.8V
nRF52840 Express	7.2 mA	1.4 mA	3.8V

Table 3: HyperCam Component Power Consumption.

in HyperCam, no library code is needed as it exclusively uses hardware-native operations such as addition, exclusive or, and comparison. Rather, HyperCam's HD classifier only requires hypervectors stored as byte arrays. This aspect not only minimizes memory requirements but also eases the process of adding new classifiers; each addition of class consumes n bytes. In contrast, a new model in DNN means its model graph (100 KB) and the weights that can be anywhere from hundreds of bytes to megabytes.

Latency Analysis. Fig. 8 shows the latency profiling of HyperCam when using Count Sketch and Bloom Filter. Observe that Bloom Filter outperforms Count Sketch for `bind & sum`, leading to an overall latency improvement. For the count-sketch method, the major processing time of `bind & sum` is not binding and summing themselves but quantizing integer values to binary for binding. This process is omitted from the Bloom Filter backend, greatly reducing latency. Additionally, in both cases, bundling does not appear in Fig. 8 because it occurs in two stages: summation as part of `bind & sum` and `majority vote`. That is, the result of binding is summed to the output element-wise and later evaluated for the majority.

6.3 Power Analysis

The power consumption of the wireless camera platform was evaluated using a Joulescope JS220 with 0.5 nA resolution, equivalent to 34 bits of dynamic range [19]. The average quiescent currents of the remaining components, the 3.3V STM32U585I and 2.8V Himax HM01B0 camera regulator, are outlined in Table 3. Next, the total power consumption of the system was evaluated during image capture, processing, and data transmission as shown in Fig. 9. The system was divided into its key constituent components during the power consumption measurement (MCU, camera, BLE module). The remaining B-U585I-IOT02A power can be derived by subtracting the power from the system's primary devices. When an event trigger occurs, the image sensor is activated, and the power consumption jumps to an average of 128 mW for

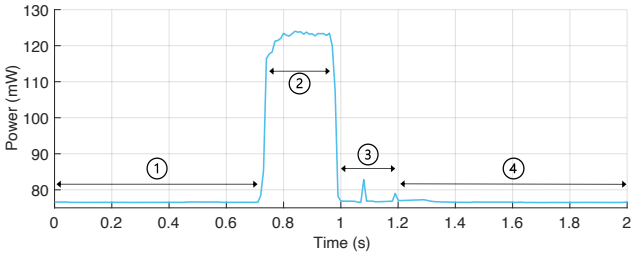

Figure 9: System Power Consumption. (1) camera platform in sleep mode (2) camera initialization, image capture, and inference, (3) data transmission, and (4) system returns to sleep mode.

image processing for 250 ms. Lastly, a data packet is transmitted to a base station over BLE for 200 ms. This equates to an average power consumption of 102 mW during active mode for a total duration of 450 ms. Note that further power optimization could be done by replacing the evaluation board's regulators with more efficient switching or LDO regulators.

7 RELATED WORK

There has been recent work in energy-efficient cameras and machine learning to exploit the resource-constrained environments presented by IoT devices. Several paradigms govern the current space, including work to ease the load of transmitting image data and onboard computing.

Onboard Computing. There have been two lines of efforts to enable IoT device onboard computing. The first is the *TinyML* paradigm, where lightweight DNNs are developed for resource-constrained platforms. It uses frameworks such as TensorFlow Lite for Microcontrollers (TFLM) to quantize and prune weights to alleviate performance overhead, enabling various levels of hardware acceleration and model deployment [10, 29]. Works such as [3, 26] use extensive network architecture search (NAS) to find an optimized neural network architecture for available memory resources. In [28], convolutional neural networks enable tiny on-device training with considerable memory limitations. Here, inference is performed, and over time, classifier weights are updated to improve performance with new input sensor data. All of these models provide concrete baselines for HyperCam's performance, as compared in Section 6.2.

On the other hand, there have been works exploring the use of hyperdimensional computing for onboard machine learning. However, the space of hyperdimensional computing primarily focuses on time-series data, which does not have the same overhead as image processing [22]. An exception to this is [15], where HDC is used to enable emotion and face detection. This work uses histograms of gradients (HoGs) to extract the features of the image, which involve

System	Active Power	Resolution	Frame Rate	Communication	Onboard Inference
BackCam [18]	9.7 mW	160×120	1 fps	backscatter	×
WISPCam [31]	6 mW	176×144	0.001 fps	backscatter	×
NeuriCam [37]	85 mW	640×480/740p	15 fps	BLE	×
MCUNet [27]	N/A	224×224	N/A	N/A	✓
HyperCam	128 mW	160×120	8 fps	BLE	✓

Table 4: IoT Camera Platforms. A comparison of HyperCam to existing camera platforms.

calculating the gradients of the pixels and binning them by angles. This front-end feature extractor not only introduces more computation load but also fails to resolve the inherent encoding complexity in HD image processing.

Energy-efficient Wireless Cameras. The advancements in low-power processors and image sensors have led to several recent works that focus on developing low-power wireless camera platforms for computer vision applications. In [30, 31], a battery-free RFID camera is presented and evaluated for machine vision applications such as face detection. Here, subsampled images are transmitted to a base station that runs a face detection algorithm. If a face is detected, coordinates of windows within the image frame are transmitted back to WISPCam to retrieve higher-resolution images. In [18], a low-power wireless camera platform is presented to support real-time vision applications where images are sent to a base station to perform image processing and face classification. Here, image compression is performed to help minimize overall latency and, in turn, reduce power consumption. More recently, [37] presents a deep-learning-based system for video capture from a low-power wireless camera platform. Similar to the aforementioned related work, the neural network processing runs at an edge server or in the cloud. Compared to prior work, HyperCam focuses on enabling onboard computer vision for energy-constrained wireless camera platforms rather than offloading image processing and classification to the edge or cloud. More closely related to HyperCam is [26], which deploys a lightweight inference engine on an MCU system for onboard image processing. Table 4 compares HyperCam to similar IoT camera platforms. Other work includes [1, 16], which present ultra-low-power implementations of wireless camera platforms for extremely challenging environments such as underwater imaging and placing wireless cameras on insects.

8 CONCLUSION AND DISCUSSION

We introduce HyperCam, an onboard image classification pipeline that leverages hyperdimensional computing (HDC). To meet the stringent resource constraints of off-the-shelf MCUs, we propose original image encoding methods involving sparse binary vectors and on-the-fly codebook generations, which significantly reduce the number of operations and memory footprint. As a result, our system requires only about 60 KB of flash memory and 20 KB of RAM, achieving a latency of approximately 0.1 s, while maintaining comparable accuracy across multiple classification tasks.

A key advantages of HyperCam is its scalability and compatibility across a wide range of hardware platforms. Unlike most ML models that rely heavily on floating-point operations and require specialized hardware support, such as Neural Processing Units (NPUs) and Floating Point Units (FPUs), HyperCam only uses bits and bit operations. Moreover, HyperCam does not require any additional libraries, machine learning engines, Neural Architecture Search (NAS), or hardware-specific optimizations to reproduce results. That is, HyperCam can be easily ported to other families of MCUs.

We highlight several future research directions:

Online Learning. Data-driven models deployed in the real world must be able to reuse new data to refine themselves. This is crucial because the deployment data can vary greatly from the training data distributions and unseen categories of data can appear. The system can adapt to these changes through online learning or incremental learning, which involves integrating new data streams during deployment. HDC offers an easy transition to online learning due to the ease of updating class hypervectors. New image hypervectors can be bundled to individual class hypervectors.

Cloud Connection. Currently, HyperCam connects to a gateway where the prediction result is transmitted and displayed. If the gateway establishes a connection to the cloud and pushes the data, remote users can access the data and monitor the results. The cloud can potentially store the bundling of thousands of image hypervectors as a model summary. This can be used to refine the model and provide future updates.

Multi-modal Sensors. Modern IoT devices are equipped with a wide array of sensors. Providing multiple sensor data inputs to one intelligent model can lead to more accurate and quick decisions without the need to fuse this data with arbitrary algorithms. HD classifiers can easily fuse multiple types of sensor data because different domains of data are all encoded to hypervectors. Examples of modalities include images and audio for speech recognition, EEG and heart rate data for anomaly detection.

Diverse Applications. HyperCam can be applied to many different computer vision tasks. Many IoT camera systems target remote environments where power and connectivity are limited, such as large-scale farms and underwater ocean profiling [1, 36]. An energy-efficient onboard classifier can run in these environments and perform tasks such as pest detection, crop yield prediction, and wildlife monitoring. A large-scale deployment of HyperCam is yet to be tested.

REFERENCES

[1] Sayed Saad Afzal, Waleed Akbar, Osvy Rodriguez, Mario Doumet, Unsoo Ha, Reza Ghaffarivardavagh, and Fadel Adib. 2022. Battery-free wireless imaging of underwater environments. *Nature Communications* 13 (2022), 5546. https://doi.org/10.1038/s41467-022-33223-x

[2] Gwangbin Bae, Martin de La Gorce, Tadas Baltrušaitis, Charlie Hewitt, Dong Chen, Julien Valentin, Roberto Cipolla, and Jingjing Shen. 2023. DigiFace-1M: 1 Million Digital Face Images for Face Recognition. In *2023 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE.

[3] Colby R. Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas Navarro, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul N. Whatmough. 2020. MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers. *CoRR* abs/2010.11267 (2020). arXiv:2010.11267 https://arxiv.org/abs/2010.11267

[4] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. 2018. Scalable methods for 8-bit training of neural networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 5151–5159.

[5] G. Benelli, G. Meoni, and L. Fanucci. 2018. A low power keyword spotting algorithm for memory constrained embedded systems. In *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. 267–272. https://doi.org/10.1109/VLSI-SoC.2018.8644728

[6] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[7] Qiong Cao, Li Shen, Weidi Xie, Omkar M Parkhi, and Andrew Zisserman. 2018. Vggface2: A dataset for recognising faces across pose and age. In *2018 13th IEEE international conference on automatic face & gesture recognition (FG 2018)*. IEEE, 67–74.

[8] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*. Springer, 693–703.

[9] Kenneth L Clarkson, Shashanka Ubaru, and Elizabeth Yang. 2023. Capacity analysis of vector symbolic architectures. *arXiv preprint arXiv:2301.10352* (2023).

[10] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. 2021. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems* 3 (2021), 800–811.

[11] M. Giordano, P. Mayer, and M. Magno. 2020. A Battery-Free Long-Range Wireless Smart Camera for Face Detection. In *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems* (Virtual Event, Japan) *(ENSsys '20)*. Association for Computing Machinery, New York, NY, USA, 29–35. https://doi.org/10.1145/3417308.3430273

[12] Alejandro Hernández-Cano, Namiko Matsumoto, Eric Ping, and Mohsen Imani. 2021. OnlineHD: Robust, Efficient, and Single-Pass Online Learning Using Hyperdimensional System. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 56–61. https://doi.org/10.23919/DATE51398.2021.9474107

[13] Himax. 2024. HM01B0 Ultralow Power CIS. https://www.himax.com.tw/products/cmos-image-sensor/always-on-vision-sensors/hm01b0/

[14] Mohsen Imani, Abbas Rahimi, Deqian Kong, Tajana Rosing, and Jan M Rabaey. 2017. Exploring hyperdimensional associative memory. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 445–456. https://doi.org/10.1109/HPCA.2017.28

[15] Mohsen Imani, Ali Zakeri, Hanning Chen, TaeHyun Kim, Prathyush Poduval, Hyunsei Lee, Yeseong Kim, Elaheh Sadredini, and Farhad Imani. 2022. Neural computation for robust and holographic face detection. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 31–36. https://doi.org/10.1145/3489517.3530653

[16] Vikram Iyer, Ali Najafi, Johannes James, Sawyer Fuller, and Shyamnath Gollakota. 2020. Wireless steerable vision for live insects and insect-scale robots. *Science robotics* 5, 44 (2020), eabb0839.

[17] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1712.05877* (2017).

[18] Colleen Josephson, Lei Yang, Pengyu Zhang, and Sachin Katti. 2019. Wireless computer vision using commodity radios. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*. 229–240.

[19] Joulescope. 2024. Joulescope JS220: Precision Energy Analyzer. https://www.joulescope.com/products/js220-joulescope-precision-energy-analyzer

[20] Pentti Kanerva. 1997. Fully distributed representation. *PAT* 1, 5 (1997), 10000.

[21] Denis Kleyko, Mike Davies, Edward Paxon Frady, Pentti Kanerva, Spencer J Kent, Bruno A Olshausen, Evgeny Osipov, Jan M Rabaey, Dmitri A Rachkovskij, Abbas Rahimi, et al. 2022. Vector symbolic architectures as a computing framework for emerging hardware. *Proc. IEEE* 110, 10 (2022), 1538–1571.

[22] D. Kleyko, D. A. Rachkovskij, E. Osipov, and A. Rahimi. 2022. A Survey on Hyperdimensional Computing Aka Vector Symbolic Architectures, Part I: Models and Data Transformations. *ACM Comput. Surv.* 55, 6, Article 130 (dec 2022), 40 pages. https://doi.org/10.1145/3538531

[23] Denis Kleyko, Abbas Rahimi, Ross W Gayler, and Evgeny Osipov. 2020. Autoscaling bloom filter: controlling trade-off between true and false positives. *Neural Computing and Applications* 32 (2020), 3675–3684.

[24] Jovin Langenegger, Geethan Karunaratne, Michael Hersche, Luca Benini, Abu Sebastian, and Abbas Rahimi. 2023. In-memory factorization of holographic perceptual representations. *Nature Nanotechnology* 18, 5 (2023), 479–485.

[25] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. 1998. The MNIST Database of Handwritten Digits. http://yann.lecun.com/exdb/mnist/.

[26] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. MCUNet: tiny deep learning on IoT devices. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 982, 12 pages.

[27] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. 2020. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems* 33 (2020), 11711–11722.

[28] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-device training under 256kb memory. *Advances*

in Neural Information Processing Systems 35 (2022), 22941–22954.

[29] Erez Manor and Shlomo Greenberg. 2022. Custom hardware inference accelerator for tensorflow lite for microcontrollers. *IEEE Access* 10 (2022), 73484–73493.

[30] Saman Naderiparizi, Zerina Kapetanovic, and Joshua R Smith. 2016. Wispcam: An rf-powered smart camera for machine vision applications. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*. 19–22.

[31] Saman Naderiparizi, Aaron N Parks, Zerina Kapetanovic, Benjamin Ransford, and Joshua R Smith. 2015. WISPCam: A battery-free RFID camera. In *2015 IEEE International Conference on RFID (RFID)*. IEEE, 166–173.

[32] Dmitriy A Rachkovskiy, Sergey V Slipchenko, Ernst M Kussul, and Tatyana N Baidyk. 2005. Sparse binary distributed encoding of scalars. *Journal of Automation and Information Sciences* 37, 6 (2005).

[33] Nordic Semiconductor. 2024. nRF52840. https://www.nordicsemi.com/products/nrf52840

[34] STMicroelectronics. 2024. Discovery Kit for IoT Node with STM32U5 Series. https://www.st.com/en/evaluation-tools/b-u585i-iot02a.html

[35] STMicroelectronics. 2024. STM32U585AI. https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html?rt=db&id=DB4410

[36] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. 2017. {FarmBeats}: an {IoT} platform for {Data-Driven} agriculture. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 515–529.

[37] B. Veluri, C. Pernu, A. Saffari, J. Smith, M. Taylor, and S. Gollakota. 2023. NeuriCam: Key-Frame Video Super-Resolution and Colorization for IoT Cameras. 1–17.

[38] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv preprint arXiv:1708.07747* (2017). https://arxiv.org/abs/1708.07747

[39] Yuchen Zhao, Sayed Saad Afzal, Waleed Akbar, Osvy Rodriguez, Fan Mo, David Boyle, Fadel Adib, and Hamed Haddadi. 2022. Towards battery-free machine learning and inference in underwater environments. In *Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications* (Tempe, Arizona) *(HotMobile '22)*. Association for Computing Machinery, New York, NY, USA, 29–34. https://doi.org/10.1145/3508396.3512877