Exploiting On-device Image Classification for Energy Efficiency in Ambient-aware Systems

Shuayb Zarar, Swagath Venkataramani, Xian-Sheng Hua, Jie Liu, and Jin Li

Abstract Ambient-aware applications need to know what objects are in the environment. Although video data contains this information, analyzing it is a challenge *esp.* on portable devices that are constrained in energy and storage. A naïve solution is to sample and stream video to the cloud, where advanced algorithms can be used for analysis. However, this increases communication-energy costs, making this approach impractical. In this article, we show how to reduce energy in such systems by employing simple on-device computations. In particular, we use a low-complexity feature-based image classifier to filter out unnecessary frames from video. To lower the processing energy and sustain a high throughput, we propose a hierarchically-pipelined hardware architecture for the image classifier. Based on synthesis results from an ASIC in a 45 nm SOI process, we demonstrate that the classifier can achieve minimum-energy operation at a frame rate of 12 fps, while consuming only 3 mJ of energy per frame. Using a prototype system, we estimate about 70% reduction in communication energy when 5% of frames are interesting in a video stream.

1 Introduction

Portable devices connect to the physical world through sensors. One rich sensing modality is the visual light field, which is captured by cameras. It provides us information about various things and events around us. Thus, perceiving the environment through a stream of video has the potential to light up a host of new context-aware applications on portable devices. Fig. 1 illustrates three such examples. First, an on-board camera can help a flying drone detect the presence of obstacles and aid in navigation [4]. Second, a dash-mounted camera can provide real-time driver assis-

Mohammed Shoaib, Xian-Sheng Hua, Jie Liu, Jin Li
Microsoft Research, Redmond WA 98052 e-mail: {moshoaib,xshua,liuj,jinl}@microsoft.com

Swagath Venkataramani
School of ECE, Purdue University, W. Lafayette IN 47907 e-mail: venkata0@purdue.edu

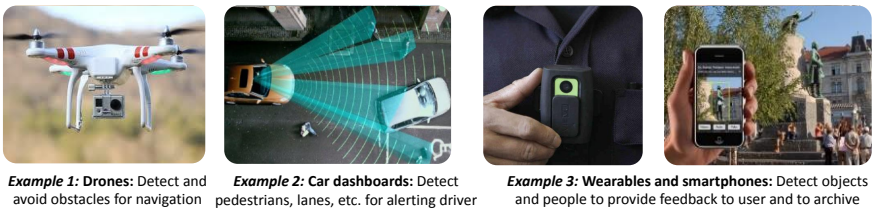


Fig. 1 Video processing can enable a range of ambient-aware applications on portable devices.

tance by identifying traffic signs, pedestrians, lanes, and other automobiles [10, 7]. Third, wearable cameras and smartphones can detect people and objects in front of them, which can help improve service and productivity [1, 5, 6, 2, 3] .

Observe that while extracting actionable information from video, a basic requirement is to detect and recognize objects in each frame. Then comes higher-level image understanding such as actions, events, *etc.* Fortunately, all three of these are rich areas of research and the literature provides many algorithmic options to solve them [9, 8, 11]. However, when realizing these techniques in an end-to-end system for portable devices, there are some new trade-offs that we need to make. We discuss some of these next.

1.1 System-level Challenges

Fig. 2 shows a block diagram of the various steps involved in realizing an ambient-aware system. It comprises computations for object detection, recognition, and image understanding. Information derived from image understanding is used to drive ambient-aware applications such as the ones described in the previous section. To be realized on portable devices, such systems need to meet three key constraints. First, most applications require ambient-aware systems to respond in real-time. One way to achieve this is to keep some sensor in the system always on. For example, a dash-mounted camera has to detect pedestrians as soon as they appear so that brakes can be applied in time, if necessary. This can be achieved by either keeping the camera always on or by using a continually operating motion detector to trigger the camera. Second, these systems must have high algorithmic accuracy. This in turn implies

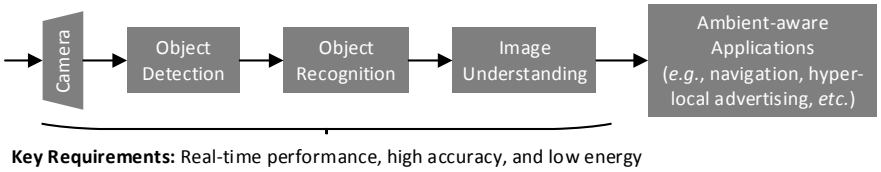


Fig. 2 An end-to-end ambient-aware system involves computations for object detection, recognition, and image understanding. To be useful in a mobile scenario, such systems need to meet strict constraints in performance and energy.

that each step in the sequence to be precise. Third, these systems must be energy-efficient when realized on a portable device. This last constraint arises due to the need for mobility in several useful ambient-aware applications.

The three system-level constraints mentioned above lead to interesting design trade-offs. Intuitively, lowering latency hints towards performing all computations locally on the portable device. However, the associated energy costs for this approach can be prohibitive. Recent evaluations with face recognition on Google Glass, an emerging wearable device, validate this behavior. Experiments show that local computations can drain the battery at a speed that is 10× faster than routine use (battery life is lowered from 377 min. to 38 min.) [12, 13]. Similar results have also been observed for other portable devices such as smartphones, drones, and security cameras [16, 17, 18]. Another trade-off is between accuracy and energy: accurate algorithms are desirable at each stage but are prohibitive on portable devices due to the high energy costs.

Since it is infeasible to support all computations locally on portable devices, there is an emerging thrust towards realizing hybrid systems. Such systems aim to exploit the growing connectivity of devices together with the computational capabilities of the cloud [14, 15]. Although promising, these hybrid systems face issues along a new dimension – they introduce additional latencies and energy costs due to data communication. Fig. 3 shows the costs involved in acquiring 3-channel RGB video [at 30 frames-per-second (fps), 2×8b per pixel] and streaming it to the cloud for processing. For the analysis shown, we assume 90 mW power for sensing 1080p/60 fps video and 240 mW for MPEG compression by 10× [21, 20]. We also assume that the power scales with the frame rate and resolution. Further, for communication using the WiFi 802.11 a/g/n protocol, we assume transmission energies of 40 nJ/b and 10 nJ/b at speeds of 54 and 150 Mbps, respectively [19]. Under these assumptions, for a portable device with a Li-ion battery of capacity 500 mAh (6660 J at 3.7 V), the streaming system model allows operation for only 96 minutes before requiring a recharge. The recharge time reduces to 78 minutes and 35 minutes for 720p and 1080p HD image resolutions, respectively. Thus, acquiring raw video on the portable device and streaming it to the cloud for processing is undesirable for continuous operation. Thus, there is a need to dissect the sequence of computations so that some are performed locally on the device and some on the cloud. Our proposed system model is guided by this insight. We present details about it next.

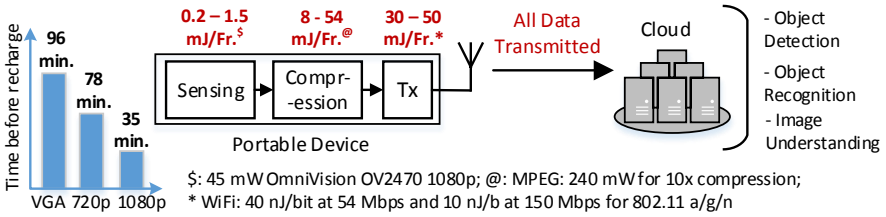


Fig. 3 Realizing an end-to-end ambient-aware system through continuous video streaming is infeasible on portable devices.

1.2 Design Approach

As an alternative to performing all computations in the cloud, we propose to split the sequence so that computations are supported in parts on the device and the cloud. In this section, we present the analysis behind our approach.

Consider the CamVid dataset, which is representative of typical recordings from a portable device [22]. Specifically, the dataset provides multiple recordings from a dash-mounted camera on a car; for illustration purposes, we have randomly chosen one recording, seq05VD, of 3 min. There are many objects of interest in the video recording. Observe from Fig. 4 that the frames-of-interest (FoI) (*i.e.*, those that contain relevant objects) comprise only a small percentage of all frames. On average, across all objects, only 10% of the frames are interesting at 10 fps. At a lower frame rate of 1 fps, this number is reduced to about 1%. This result shows that just after the object detection step, the amount of useful data (determined by FoI) can be reduced by 90–99%. Processing through the object-recognition step can further lower the number of informative frames. However, the room for improvement due to this step is low. Thus, in our end-to-end system, we propose to employ computations for object detection (used synonymously with image classification) locally on the portable device, while performing all other computations in the cloud. Through this approach, we will demonstrate that we can substantially reduce the amount of communication energy (and thus the end-to-end system energy). To keep the image-classification energy low, we will also show that we need to subtly tweak the algorithmic accuracy as well as develop a dedicated hardware accelerator.

Our system model is shown in Fig. 5. Under the same assumptions as those used for Fig. 3, we observe that using a local data filter for image classification on the portable device can improve battery lives by up to 5.5× (*i.e.*, battery life improves from 96 min. or 1.6 hrs. in Fig. 3 to 8.8 hrs. in our case for VGA frames). These energy savings come due to a reduction in the communication energy. Observe that in estimating the gains, we assume that the local filter for image classification reduces useful data frames by 90% and that it costs an additional 3 mJ/frame. Next, we vali-

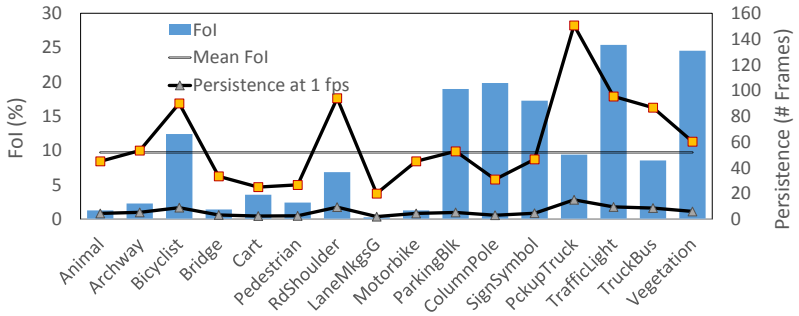


Fig. 4 Results from a typical video dataset show that most object persist in the camera's field of view for at least 10 frames. In a recording of approx. 3 min., on average, specific objects appear in $\leq$ 10% of the frames at 10 fps and in about 1% of the frames at 1 fps.

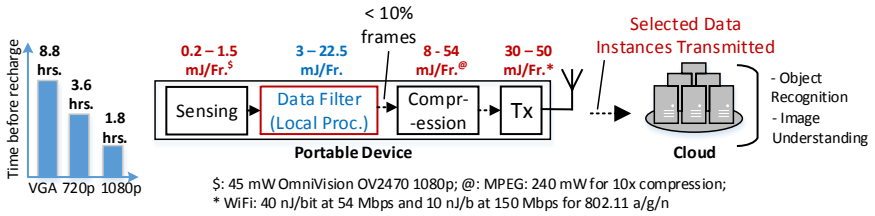


Fig. 5 Proposed system model: Perform object detection locally on the device. This approach can increase battery lives by up to 5.5× (*i.e.*, 96 min. in Fig. 3 to 8.8 hrs. in our case for VGA frames).

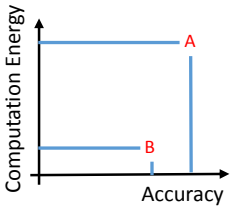
date these assumptions and describe the trade-offs that exists between accuracy and energy consumption of the data filter.

2 Algorithm Selection for Data Filtering

Recall from Fig. 4 that the FoI reduces with frame rate. Also, note from the figure that once an object is found in a frame, it stays in the camera's field of view for at least 5-10 subsequent frames, when the video is sampled at 30 fps. We call this behavior *persistence*. The value of persistence is shown for the various objects in the CamVid recording on the secondary Y-axis in Fig. 4. This high value of persistence hints at the fact that we could lower the frame rate by 5-10× and still detect the presence of interesting objects in the video. Equivalently, we could relax the accuracy of the image classification algorithm so that it detects at least one out of the 5-10 contiguous frames in which the object of interest appears. In our system, we propose to exploit a combination of both of these approaches.

To sustain the battery charge up to a reasonably long duration, we assume a computational energy budget of approx. 3-20 mJ (Fig. 5), depending on the image resolution. Assuming a 50 mW budget for VGA (lowest) resolution, this translates to 17 fps, 100 million operations per second (MOPS) [costing 2 mJ/Fr. total, assuming 0.3 μW/OP], and less than 10 MB of memory accesses [costing 1 mJ/Fr. total, assuming 100 pJ/B access energy] per frame. Thus, our energy budget still allows room for relaxing the accuracy of the algorithm. We achieve this by employing the technique of biased classifiers. We explain this concept next.

Fig. 6, in the middle, shows the energy constraints for implementing the detection algorithm locally on the portable device. At the left, the figure also shows two potential algorithm choices that we have for implementing image classification, namely, A and B. Algorithm A has high accuracy but also high computational energy. Algorithm B, on the other hand, has both lower accuracy and energy. On the right, the figure shows two metrics that represent the accuracy of the algorithms, namely, true positives and false positives. Observe how for algorithm B one metric is lower and another higher than algorithm A. True positives are determined by the number of frames transmitted (FT) (or selected) by the algorithm that are among the FoI – it is desirable to have these high. False positives are determined by the



Target Specifications	
Complexity	<100 MOPs
Memory	<10 MB
Energy	<10 mJ/Frame
Power	<50 mW
Latency	Real time
% Data Reduction	High

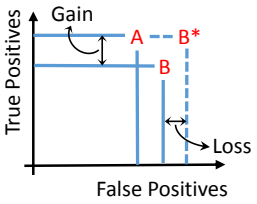


Fig. 6 We propose to bias a low-energy algorithm B towards having high true positives at the cost of additional false positives (resulting in an algorithm B*). Gain and loss are annotated for algorithm B* in comparison with algorithm B.

FT that are not among the FoI – it is desirable to have these low. Typically, both of these metrics are related to each other, increasing (decreasing) one also increases (decreases) the other. But, the change is not symmetric. In other words, increasing the true positives by x% does not necessarily increase the false positives by the same percentage. In fact, the change is dependent on the algorithm at hand. We propose to exploit this niche property of classification algorithms in tweaking the on-device image classifier.

Our proposal is to bias algorithm B such that it leads us to a new algorithm B*, which has a high true positive rate (potentially close to that provided by algorithm A) at the cost of a higher false positive rate than algorithm B (and algorithm A). For ambient-aware applications, having high true positives is important since the algorithm then does not miss frames that contain objects of interest. The above process thus implies that algorithm B* transmits a few additional frames (comprising of the additional false positives) when compared to algorithm A but is able to detect all of the interesting frames that algorithm A would detect. However, an important point to note is that this higher false positive rate of B* comes with an energy benefit over A – recall that the energy requirements of algorithm B (and thus also B*) were much lower than algorithm A to begin with.

The amount of energy algorithm B* helps us save end-to-end depends on how simple algorithm B* is in comparison to algorithm A. Consider the computational energy costs for algorithm B* ranging from 5-40 mJ/Fr. Fig. 7 shows the end-to-end energy savings that are achievable with these potential costs. If algorithm B* costs 40mJ/Fr. for image classification, end-to-end energy savings are achieved only until the number of frames transmitted (%FT) is $\leq$ 40%. Thus, if %FoI is 10%, there is an additional room of 30% for the increasing false positive rate. However, if algorithm B* costs only 5 mJ/Fr. then end-to-end energy savings are achieved until 94%, resulting in a room of 84% for the increase in false positive rate. Thus, to maximize the end-to-end energy savings, it makes more sense to choose an algorithm B* that is energy efficient and has a higher false positive rate (like algorithm B*) than one that has higher energy costs and a lower false positive rate (like algorithm A). The image-classification algorithm that we select for our system is based on this principle. We present details about it next.

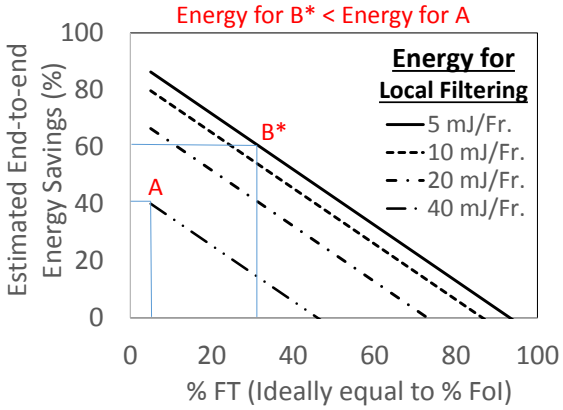


Fig. 7 Since algorithm B* has much lower computational energy costs, it provides us higher end-to-end energy savings than algorithm A. Figure adapted from [25].

3 Low-energy Algorithm for Image Classification

Recent results have shown that neural network based algorithms have the potential to provide state-of-the-art accuracy in image classification as well as in visual recognition [23]. These algorithms employ dynamic decision models that require large memories, high-bandwidth communication links, and compute capacities of up to several GOPS [43, 42, 41]. With enormous potential parallelism, such algorithms provide very high accuracies. However, these algorithms are not suited for implementation in our case. This is because, as mentioned earlier, our goal is not to select the algorithm with the highest accuracy but the one with the lowest energy consumption. It is also desirable that the algorithm that we choose be programmable so that it can detect arbitrary objects of interest. We thus choose an algorithm that not only performed reasonably well in the ILSVRC competition, but also that which had a much lower computational complexity [24]. The basic algorithm is illustrated in Fig. 8. It comprises four major computational blocks that we describe next.

3.1 Interest-point Detection (IPD)

For each incoming frame, this step helps identify the pixel locations with the most information. Locations typically lie at key-points such as corners, edges, blobs, ridges, *etc.* In our case, we utilize the Harris-Stephens algorithm that detects pixel locations on object corners [31]. In this algorithm, a patch of pixels $I(x,y)$ is extracted around each pixel location (x,y) in a grayscale frame I. This patch is subtracted from a shifted patch $I(x+u,y+v)$ centered at location $(x+u,y+v)$ and the result is used to compute the sum-of-squared distances [denoted by $S(x,y)$] using the following formulation:

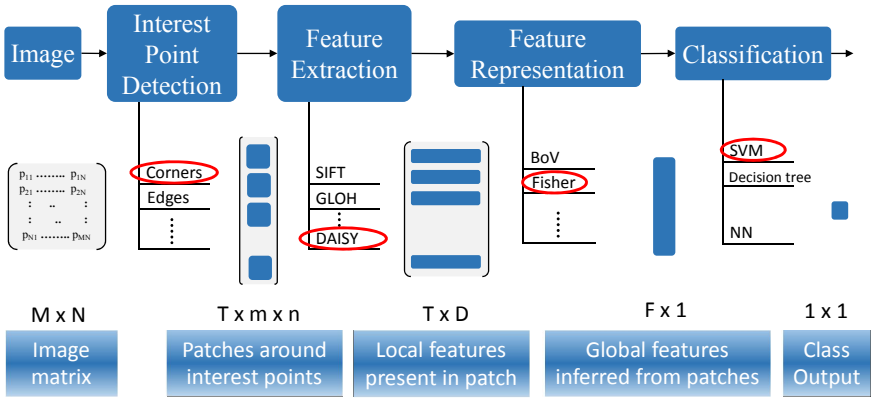


Fig. 8 Light-weight algorithm used for image-classification on the portable device. At each stage, our selection is shown circled and the dimensionality of data is shown at the bottom.

$$S(x,y) = \Sigma_u \Sigma_v w(u,v)[I(u+x,v+y) - I(u,v)]^2, \tag{1}$$

where $w(u,v)$ is a window function (matrix) that contains the set of weights for each pixel in the frame patch. The weight matrix could comprise a circular window of Gaussian (isotropic response) or uniform values. In our case, we pick uniform values since it simplifies implementation. A corner is then characterized by a large variation of $S(x,y)$ in all directions around the pixel at (x,y). In order to aid the computation of $S(x,y)$, the algorithm exploits a Taylor series expansion of $I(u+x,v+y)$ as follows:

$$I(u+x,v+y) \approx I(u,v) + I_x(u,v)x + I_y(u,v)y \tag{2}$$

where $I_x(u,v)$ and $I_y(u,v)$ are the partial derivatives of the image patch I at (u,v) along the x and y directions, respectively. Based on this approximation, we can write $S(x,y)$ as follows:

$$S(x,y) \approx \Sigma_u \Sigma_v w(u,v) \cdot [I_x(u,v) \cdot x - I_y(u,v) \cdot y]^2 \;\approx\; [x,y]\; A\; [x,y]^T \tag{3}$$

where A is a structure tensor that is given by the following:

$$\begin{vmatrix} < I_x^2 > & < I_x I_y > \\ < I_x I_y > & < I_y^2 > \end{vmatrix}. \tag{4}$$

In order to conclude that (x,y) is a corner location, we need to compute the eigenvalues of A. But, since the exact computation of the eigenvalues is computationally expensive, we can compute the following corner measure $M_{c'}(x,y)$ that approximates the characterization function based on the eigenvalues of A:

$$M_{c'}(x,y) = det(A) - \kappa \cdot trace^2(A). \tag{5}$$

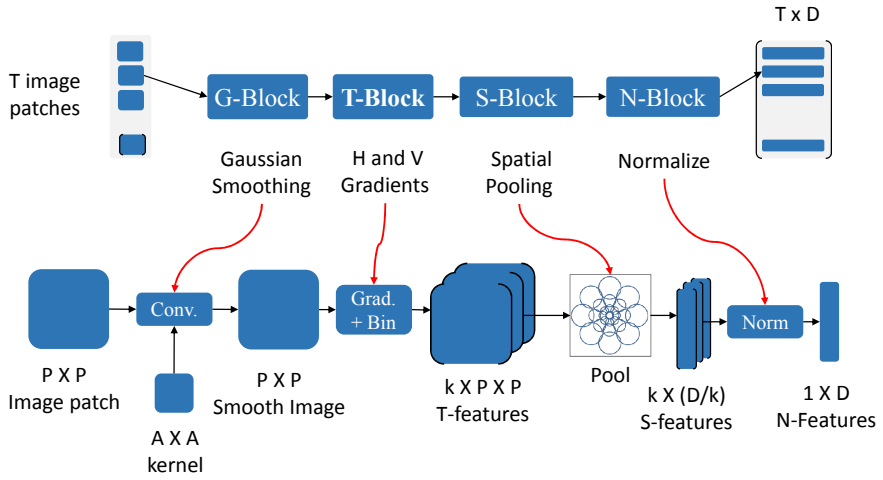


Fig. 9 We use the daisy feature extraction algorithm. It comprises T, S, N, and E processing blocks.

To be more efficient, we avoid setting the parameter κ and make use of a modified corner measure $M_c(x,y)$, which amounts to evaluating the harmonic mean of the eigenvalues as follows:

$$M_c(x,y) = 2 \cdot det(A) / [trace(A) + \varepsilon] \tag{6}$$

where ε is a small arbitrary positive constant (that is used to avoid division by zero). After computing a corner measure $[M_c(x,y)]$ at each pixel location (x,y) in the frame, we need to assess if is largest among all abutting pixels and if it is above a pre-specified threshold; marking it to be a corner if it is. This process is called non-maximum suppression (NMS). The corners thus detected are invariant to lighting, translation, and rotation.

3.2 Feature Extraction

The feature-extraction step extracts low-level features from pixels around the interest points. Typical classification algorithms use histogram-based feature-extraction methods such as SIFT, HoG, GLOH, *etc.* While appearing quite different, many of these can be constructed using a common modular framework consisting of five processing stages, namely G-block, T-Block, S-Block, E-Block, and N-Block [27, 33]. This approach known as the daisy feature-extraction algorithm, thus allows us to adapt one computation engine to represent most other feature-extraction methods depending on tunable algorithmic parameters that can be set at run-time. Fig. 9 shows a block-level diagram of the daisy feature-extraction module. At each stage, different candidate block algorithms may be swapped in and out to produce new overall descriptors. In addition, parameters that are internal to the candidate features

can be tuned in order to maximize the performance of the descriptor as a whole. We next present details about each of the processing stages.

- ***Pre-smoothing (G-block):*** A $P \times P$ patch of pixels around each interest point is smoothed by convolving it with a 2d-Gaussian filter of standard deviation (σ_s).
- ***Transformation (T-block):*** This block maps the smoothed patch onto a length k vector with non-negative elements. There are four sub-blocks defined for the transformation, namely, T1, T2, T3, and T4. In our system, we have implemented only the T1 and T2 blocks, with easy extensibility options for T3 and T4.
  - **T1:** At each pixel location (x,y), we compute gradients along both horizontal (Δx) and vertical (Δy) directions. We then apportion the magnitude of the gradient vector into k (equals 4 in T1a and 8 in T1b mode) bins split equally along the radial direction – resulting in an output array of k feature maps, each of size $P \times P$.
  - **T2:** The gradient vector is quantized in a sine-weighted fashion into 4 (T2a) or 8 (T2b) bins. For T2a, the quantization is done as follows: $|\Delta_x| - \Delta_x$; $|\Delta_x| + \Delta_x$; $|\Delta_y| - \Delta_y$; $|\Delta_y| + \Delta_y$. For T2b, the quantization is done by concatenating an additional length 4 vector using Δ_{45}, which is the gradient vector rotated through 45^o.
  - **T3:** At each pixel location (x,y), we apply steerable filters using n orientations and compute the response from quadrature pairs. After this, we quantize the result in a manner similar to T2a to produce a vector of length $k = 4n$ (T3a) and T2b to produce a vector of length $k = 8n$ (T3b). It is also possible that we use filters of second or higher-order derivatives and/or broader scales and orientations in combination with the different quantization functions.
  - **T4:** We compute two isotropic difference of Gaussian (DoG) responses with different centers and scales (effectively reusing the G-block). These two responses are used to generate a length $k = 4$ vector by rectifying the positive and negative parts into separate bins as described in T2.
- ***Spatial Pooling (S-block):*** In this stage, we accumulate weighted vectors from the previous stage to give N linearly summed vectors of length k. This process is similar to the histogram approach used other descriptor algorithms in the literature. We concatenate these N vectors to produce a descriptor of length kN. Fig. 10 shows an overview of the different approaches. We use the following pooling patterns for the vectors:
  - **S1:** Square grid of pooling centers. The overall footprint of this grid is a parameter. The T-block features are spatially pooled by linearly weighting them according to their distances from the pooling centers.
  - **S2:** This is similar to the spatial histogram used in GLOH [34]. We use a polar arrangement of summing regions. The radii of the centers, their locations, the number of rings, and the number of locations per angular segment are all parameters that can be adjusted (zero, 4, or 8) to maximize performance.

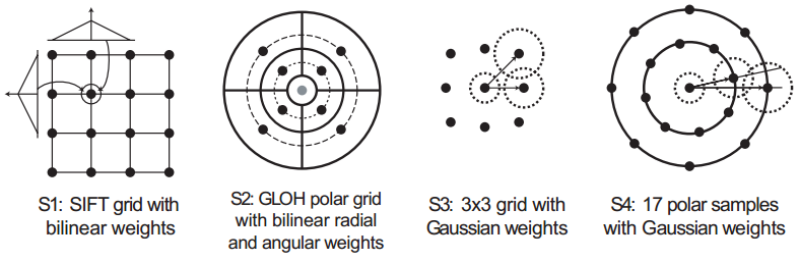


Fig. 10 Examples of the various spatial summation patterns. Figure reproduced from [32].

  - **S3:** We use normalized Gaussian weighting functions to sum input regions over local pooling centers arranged in a 3×3, 4×4, or 5×5 grid. The sizes and the positions of these grid samples are tunable parameters.
  - **S4:** This is the same approach as S3 but with a polar arrangement of the Gaussian pooling centers instead of being rectangular. We used 17 or 25 centers with the ring sizes and locations being tunable parameters.

- ***Embedding (E-block):*** This is an optional stage that is mainly used to reduce the feature vector dimensionality. This comprises multiple sub-stages: principal component analysis (E1), locality preserving projections (E2) [35], locally discriminative embedding (E3) [36], *etc.* In our design, we have not implemented the E-block but provide an option for extensibility.
- ***Post Normalization (N-block):*** This block is used to remove descriptor dependency on image contrast. In the non-iterative process, we first normalize the s-block features to a unit vector (dividing by the Euclidean norm) and clip all elements that are above a threshold. In the iterative version of this block, we repeat these steps until a maximum number of iterations have been reached.

3.3 Feature Representation

This step allows us to aggregate feature-vectors from all image patches to produce a vector of constant dimensionality. Again, there are several algorithmic options for high-level feature representation including the bag-of-visual words, fisher vectors (FV), *etc.* [26]. We choose the FV, which is a statistical representation obtained by pooling local image features. The FV representation provides high classification performance, thanks to a richer Gaussian mixture model (GMM)-based representation of the visual vocabulary. Next, we provide a description of the FV representation.

Let $I = (x_1, x_2, \ldots, x_T)$ be a set of T feature descriptors (*i.e.*, the daisy features) extracted from an image each of dimensionality D. Let $\Theta = (\mu_k, \Sigma_k, \phi_k, k = 1, 2, \ldots, K)$ be the parameters of a GMM fitting the distribution of the daisy descriptors. The GMM associates each vector x_i to a centroid k in the mixture with a strength given by the following posterior probability:

$$q_{ik} = \frac{exp\left[-\frac{1}{2}(x_i-\mu_k)^T\Sigma_k^{-1}(x_i-\mu_k)\right]}{\Sigma_{t=1}^{K} exp\left[-\frac{1}{2}(x_i-\mu_t)^T\Sigma_k^{-1}(x_i-\mu_t)\right]}. \tag{7}$$

For each centroid k, the mean (u_{jk}) and covariance deviation (v_{jk}) vectors are defined as follows:

$$u_{jk} = \frac{1}{T\sqrt{\pi_k}}\Sigma_{i=1}^{T} q_{ik}\frac{xji-\mu jk}{\sigma_{jk}} \tag{8}$$

$$v_{jk} = \frac{1}{T\sqrt{2\pi_k}}\Sigma_{i=1}^{T} q_{ik}\left[\left(\frac{x_{ji}-\mu_{jk}}{\sigma_{jk}}\right)^2 - 1\right]. \tag{9}$$

where $j = 1,2,\ldots,D$ spans the vector dimensions. The FV of an image I is the stacking of the vectors u_k and then of the vectors v_k for each of the K centroids in the Gaussian mixtures:

$$FV(I) = [\ldots u_k \ldots v_k \ldots]^T. \tag{10}$$

To get a good classification performance, the FVs need to be normalized. This is achieved by reassigning each dimension z of an FV to be $|z|^\alpha\ sign(z)$, where α is a design parameter that is optimized to limit the dynamic range of the normalized FVs. The FVs are normalized a second time by dividing each dimension by the l^2 norm. The normalized FVs thus produced are global feature vectors of size $2KD$.

3.4 Feature Classification

To keep the computational costs low, we use a simple margin-based classifier [specifically, a support vector machine (SVM)] to classify the FVs. The classifier thus helps detect relevant frames based on a model that is learned offline using pre-labeled data during the training phase. In SVMs, a set of vectors (total N_{SV} vectors), called support vectors, determine the decision boundary. During online classification, the FV is used to compute a distance score (D_S) as follows:

$$D_S = \sum_{i=1}^{N_{SV}} K(FV \cdot sv_i)\,\alpha_i y_i - b, \tag{11}$$

where sv_i is the i^{th} support vector; b, α_i, and y_i are training parameters; and the function $K(\cdot)$ is the kernel function, which is a design parameter. In our implementation, we choose polynomial kernels (up to order 3), which are defined as follows:

$$K(FV \cdot sv_i) = (FV \cdot sv_i + \beta)^d, \tag{12}$$

where d and β are training parameters. Based on the sign of D_S, an FV is assigned to either the positive (object of interest) or the negative class. To bias the classifier towards having a high true positive rate at the cost of increased false positive rate, we modify the decision boundary using the various training parameters.

Img. Scale			Daisy T-Blk				Daisy S-Blk			SVM			Accuracy
1	2	4	T14	T24	T18	T28	Rect	1r8s	2r8s	Lin	Poly3	RBF	
X			X						X		X		0.65
X			X				X			X			0.8
X			X				X				X		0.85
X			X				X					X	0.8
X				X					X		X		0.6
X				X			X				X		0.6
X					X		X				X		0.6
X					X			X			X		0.5
X					X				X		X		0.5
X						X	X				X		0.75
X						X		X			X		0.45
X						X			X		X		0.6
	X		X				X			X			0.8
	X		**X**				**X**				**X**		**0.85**
	X		X				X					X	0.85
	X				X		X				X		0.85
	X					X	X				X		0.8
	X			X					X		X		0.6
		X	X				X				X		0.85
		X			X		X				X		0.8
		X				X	X				X		0.8

Fig. 11 Design-space exploration of the algorithmic parameters for Caltech256: The highlighted row gave the best performance and the algorithmic parameters were picked accordingly.

4 Software Implementation of On-device Image Classification

We implemented the end-to-end algorithm in C# and parallelized the code using the task parallel library (TPL) provided by the .NET 4.5 framework [37]. To evaluate the algorithm, we used the following four image-classification datasets: Caltech256 [30], NORB [28], PASCAL VOC [29], and CamVid [22]. For each of the above datasets, we performed a design-space exploration of the algorithmic parameters to determine the best-performing values. Fig. 11, for instance, summarizes the exploration results for Caltech256. The highlighted row gave the best accuracy and the algorithmic parameters were chosen accordingly. Specifically, the image scale factor was set to 2 along with T14-Rect for the daisy features and 3rd degree polynomial kernel for the SVM. We also explored other microparameters (not shown in Fig. 11) such as the number of GMM clusters and α scale values for the FVs *etc.* After finding the best-performing parameters, we biased the SVM classifier using data re-sampling so that the end-to-end algorithm has a high true positive rate. In the rest of the article, we use the following two algorithmic performance metrics: (1) *coverage*, which basically represents the true positive rate [but alludes to the FoI that are detected (or covered) by the algorithm], and (2) FT, which represents a combination of the false positives and true positives.

Fig. 12 shows the FT *vs.* FoI charts for the four datasets. Results are shown at four different coverage levels: 30-50%, 50-70%, 70-90%, and 90-100%. These coverage levels mean that the respective percentage of interesting frames are selected or detected by the algorithm. Like previously mentioned, we bias the classifier to achieve these coverage levels. The error bars shown in the figure represent the variance across different objects of interest. The dotted line along the diagonal indicates the ideal value of FT (= FoI) the different coverage levels. Note that some lines cross over the others i nthe figure. This is an artifact of our experimental data; we believe

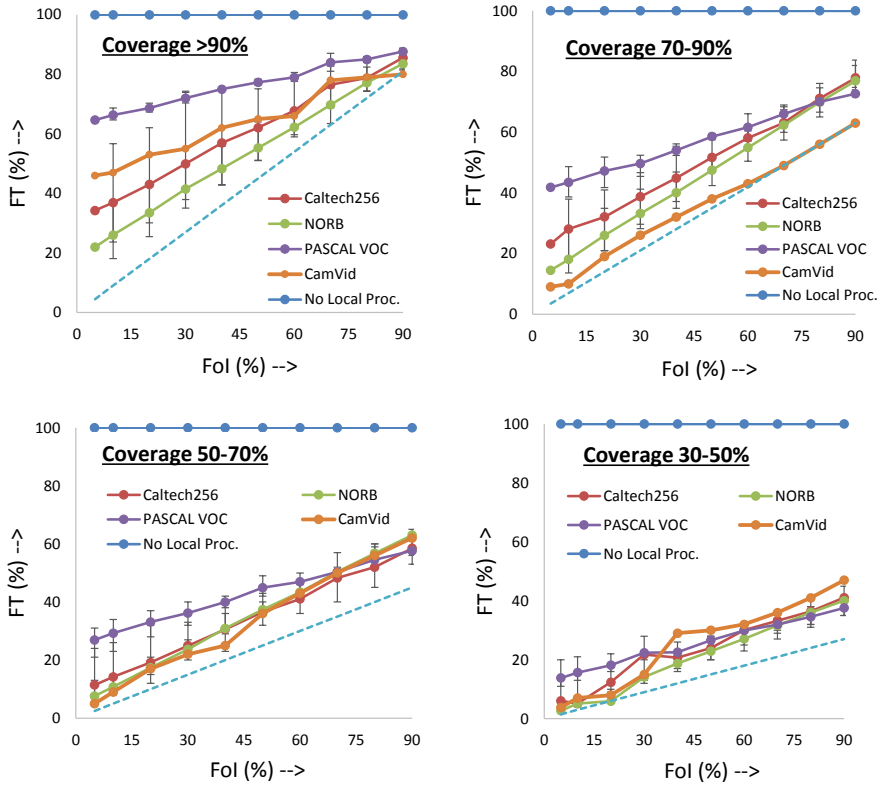


Fig. 12 FT, which is $\geq$ FoI at higher coverage values, begins to approach FoI as we relax the coverage levels of the algorithm.

that repeating the experiment for more objects (or different combinations of objects) and averaging the results would smooth the trends and remove the cross overs.

From Fig. 12, we observe that without any on-device classification, FT is always 100%; this represents the streaming system model of Fig. 3. Further, with local image classification, for a coverage of $\geq$ 90%, we are able to filter out $\sim$70% (FT = 30%) of the frames (averaged over all datasets) at FoI = 5%. This number improves dramatically at lower coverage levels (*i.e.*, goes down to 73%, 83%, and 91% at coverage levels of 70-90%, 50-70%, and 30-50%, respectively). Lower coverage levels are acceptable since typical datasets have substantial persistence (recall that persistence was 10% at 10 fps in Fig. 4).Thanks to high persistence, the probability of detecting at least one frame that contains the object of interest is thus high even at low coverage levels. The large amounts of data filtering that we achieve though local filtering translates directly into big system-level energy savings that we present ahead in Sec. 6.2.1.

Although promising from an accuracy perspective, the software implementation of the algorithm fares poorly when it comes to runtime costs. Table 1 shows how

Table 1 Software implementation of image classification incurs a large processing delay that is unacceptable for real-time context-aware applications. Table reproduced from [25].

	Caltech256	**NORB**	**PASCAL**	**CamVid**
Frame Size	640 × 480	96 × 96	640 × 480	720 × 960
MOPS	161	9	81	211
Time/frame (sec.)	3.5	0.33	1.6	4.5

the algorithmic complexity varies depending on the frame size, number of interest points, classifier model size, *etc.* (these parameters are dataset dependent). Across all datasets, we find that the mean complexity is quite low: ∼116 MOPS. However, the software run-time on both a desktop (Core i7) and mobile CPU (Snapdragon 800) exceeds 2.5 sec./frame on average. This latency comes about because we are unable to fully exploit the inherent parallelism in the algorithm. Since this latency is unacceptable for real-time context-aware applications, we propose to accelerate the image-classification algorithm through hardware specialization. We describe this approach next.

5 Hardware Implementation of On-device Image Classification

In this section, we propose a hardware-specialized engine called SAPPHIRE for accelerating image classification on portable devices. Fig. 13 shows a block diagram of the proposed architecture of a local computation platform for image classification. An ARM-class processor is used to preprocess video frames as they stream in. The raw frames are then handed off to the SAPPHIRE accelerator, which performs image classification in an energy-efficient manner. The frames selected by SAPPHIRE are then compressed by the processor and streamed out over a communication link. Within the accelerator, we exploit several microarchitectural optimizations to achieve significant processing efficiency. A key feature is that it can be configured to obtain different power and performance points for a given application. Thus, SAPPHIRE can be easily scaled to cater to both the performance constraints of the application and the energy constraints of the device. In this section, we pro-

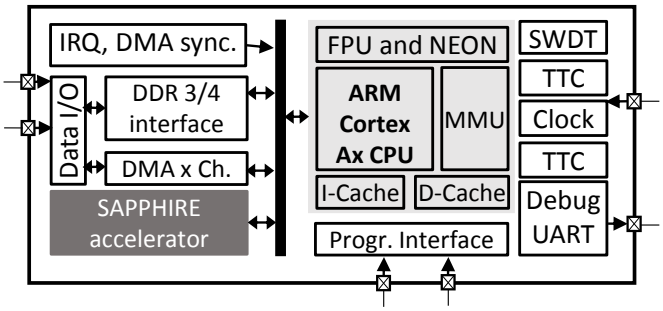


Fig. 13 Proposed use of an accelerator (SAPPHIRE) for image classification on portable devices.

vide details on the hardware optimizations that we use in SAPPHIRE followed by block-level implementations of the various computing modules that comprise the accelerator.

5.1 Hardware Optimizations

Through SAPPHIRE, we provide two key microarchitectural features: (1) stream processing support through local data buffering and two-level vector reduction, and (2) data-level parallelism through hierarchical pipelining. We describe these features next.

5.2 Stream Processing

Our proposed architecture for SAPPHIRE allows for stream processing through two techniques. First, it allows data to be buffered locally, which obviates the need for multiple fetches from external memory. Thus, the required external memory bandwidth requirements of SAPPHIRE are low. Second, we support a feature called 2-level vector reduction. This is a commonly occurring computational process in our system wherein vector data is processed in two stages. Fig. 14 illustrates the concept more generally. In the first level of reduction (*i.e.*, L1), two vectors operands U and V are processed element-wise using a reduction function f. To achieve this, we exploit inter-vector data parallelism (we provide more details about parallelism in Sec. 5.3), which enables us to reuse the vector V across all L1 lanes. Thus, the operation can be iteratively completed within a systolic array. In the second level of reduction (*i.e.*, L2), each element of the resulting vector W is processed by another reduction function g. To achieve this, we decompose U and interleave the element-wise operations. A common example of 2-level vector reduction is the computation of dot-products between two vectors in the first level followed by multiply-accumulation of the resulting vector in the second level. Thanks to 2-level vector reduction, we can avoid re-fetching data repeatedly from external memory. Thus, both memory bandwidth and local storage are significantly lowered.

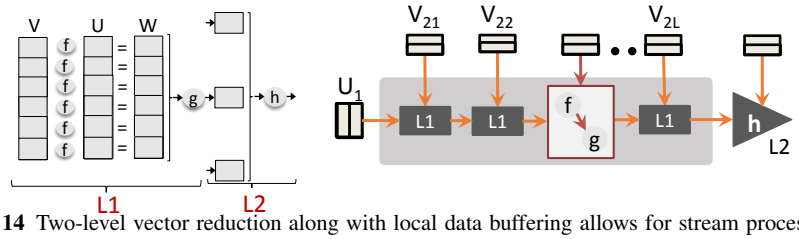


Fig. 14 Two-level vector reduction along with local data buffering allows for stream processing on SAPPHIRE.

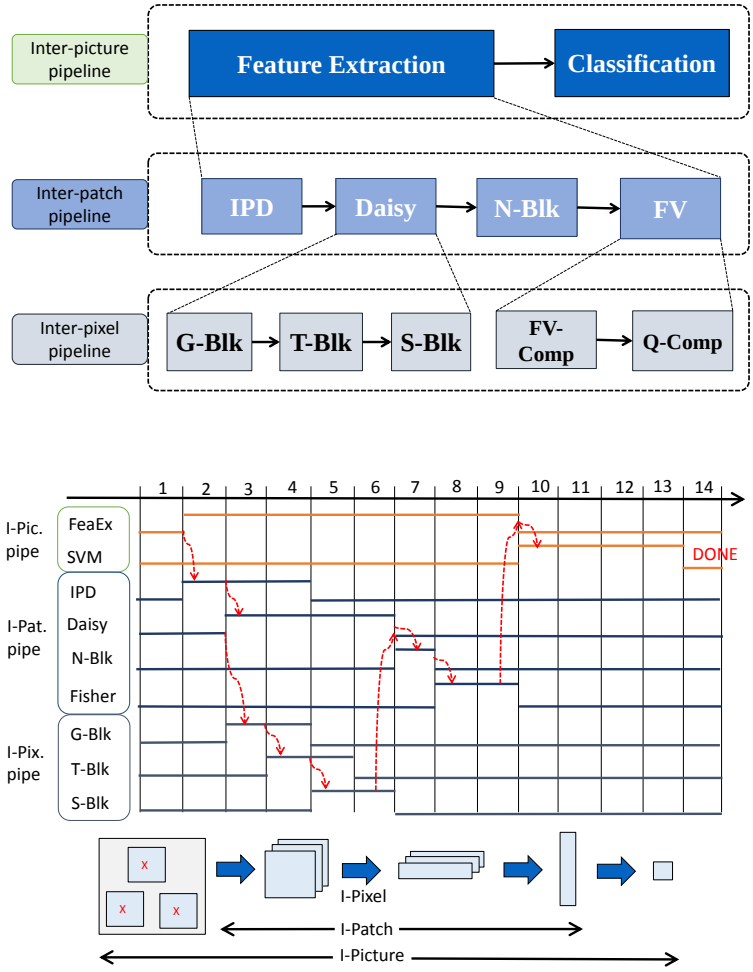


Fig. 15 Hierarchical pipelining in SAPPHIRE and the timing diagram for pipelining.

5.3 Data-level Parallelism

The image classification algorithm provides abundant opportunity for parallel processing. Since, SAPPHIRE operates on a stream of frames, it is throughput limited. Thus, we also exploit data-level parallelism through pipelining. An interesting feature of the algorithm is that the pipelined parallelism is not available at one given level, but rather buried hierarchically across multiple levels of the design. To exploit this parallelism, we develop a novel three-tiered, hierarchically pipelined architecture shown in Fig. 15. The timing diagram for hierarchical pipelining is also shown in the figure. Next, we provide details about the functional aspects of the system.

Inter-picture pipeline. This is the topmost tier in the pipeline. Here, we exploit parallelism across successive input video frames. As shown in Fig. 15, this stage comprises two parts, namely feature computation and classification. Feature computation includes IPD, daisy feature extraction, and the FV blocks. And classification comprises just the SVM. As shown in the timing diagram, while global features of a frame i are being computed, the previous frame *i.e.*, $i-1$ is concurrently processed by the classifier.

Inter-patch pipeline. This is the next tier in the pipeline. Here, we exploit parallelism within each feature-computation stage of the inter-picture pipeline. In this tier, image patches around different interest points are processed concurrently. Thus, this tier comprises the IPD, daisy (G, T, and S blocks only), and the FV modules. Interest points that are found by the IPD are pushed onto a first-in first-out (FIFO) memory, which are then utilized by the daisy sub-blocks to compute the S-block features. These features are then normalized to produce the full local descriptors at that interest point. The normalized vectors are consumed by the FV block, which iteratively updates the global feature memory. The entire process is repeated until the local memory is empty. It is interesting to note that the stages of computation in this tier cannot be merged with the previous tier since global FV computations require all descriptors (*i.e.*, descriptors at all interest points) to be available before evaluation. Due to this dependency, these tiers must be independently operated.

Inter-pixel pipeline. This is the innermost tier of the hierarchy and is present within the G, T, and S blocks of the inter-patch pipeline. It leverages the parallelism across pixels in a patch by operating them in a pipeline. The three daisy sub-blocks (*i.e.*, G, T, and S) together compute the S-Block feature output for each image patch in the frame.

To maximize throughput, it is important to balance execution cycles across all tiers of the pipeline. This, however, requires careful analysis since the execution time of each block significantly differs based on the input data and other algorithmic parameters. For instance, the delay of the second tier is proportional to the number of interest points, which varies across different video frames. Thus, in our implementation, we systematically optimize resource allocation for the various blocks based on their criticality to the overall throughput. To better understand the various inter-twined hardware-software trade-offs, we next describe the microarchitectural details of the computational block in SAPPHIRE.

5.4 Microarchitecture of Computational Blocks

In addition to pipelining, the algorithm also allows fine-grained parallel implementations within the various processing elements of SAPPHIRE. Many blocks involve a series of 2-level vector reduction operations. In our design, we employ arrays of

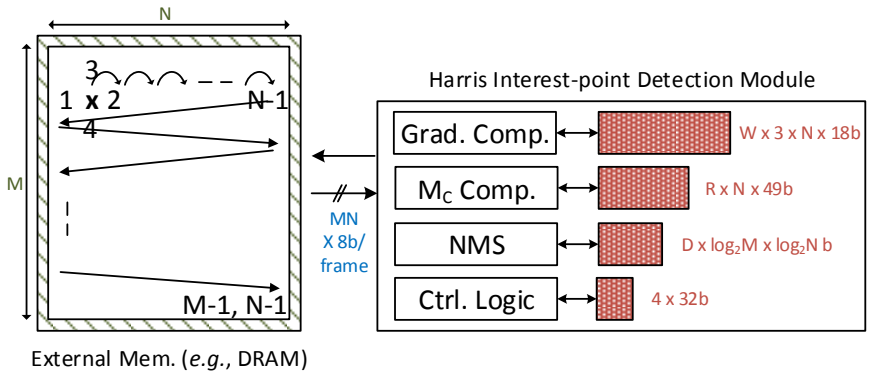


Fig. 16 Block diagram of the implemented IPD module: For typical algorithmic parameters, SAPPHIRE requires an external bandwidth of 70.31 Mbps for VGA, 0.46 Gbps for 1080p, and 1.85 Gbps for 4k image resolutions at 30fps.

specialized processing elements that are suitably interconnected to exploit this computation pattern. We also employ local buffering at various stages of processing. In this section, we describe the microarchitectural details of the different blocks in SAPPHIRE.

5.4.1 The IPD Block

A block diagram of the hardware architecture for IPD is shown in Fig. 16. For every pixel, we retrieve 4 pixels from the neighborhood using the ordering shown in the figure. The pixels are fetched from external memory (8b/pixel) using an address value that is generated by the IPD block. Thus, the external memory bandwidth required for this operation is $4MN\times$ 8b/frame, where M and N are the height and width of the grayscale frame. For VGA resolution at 30 fps, this bandwidth would be 281 Mbps and for 720p HD resolution at 60 fps, this would be 1.6 Gbps. Note that this is modest since typical DDR3 DRAMs provide a peak bandwidth of up to several 10s of Gbps.

The four abutting pixels are then used to compute the gradients along the horizontal and vertical directions, which are buffered into a local FIFO memory of size $W \times 3 \times N \times 18b$ (in a nominal implementation $W = 3$ and the memory is of size 12.7 kB for VGA and 25.3 kB for 720p HD). These gradients are in turn used to evaluate the corner measure (M_c). The data path comprises one CORDIC-based divider besides other simple compute elements. The resulting corner measures are put in a local FIFO of depth R (typically 3). This FIFO is thus of size 9.8 kB for VGA and 19.5 kB for 720p HD. The M_c values are then processed by the NMS block, which pushes the identified interest point locations (both x and y coordinates) onto another local FIFO of depth D (typically 512). Thus, the FIFO capacity is typically equal to 5.2 kB for VGA and 6.1 kB for 720p HD. In conclusion, if all pixels are accessed from external memory, the total bandwidth requirements for the IPD block

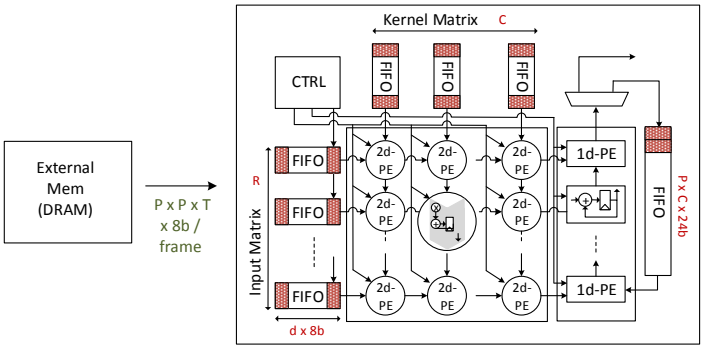


Fig. 17 Block diagram of the systolic array architecture used for 2d-convolution in the G-block.

are: 70.31 Mbps for VGA, 0.46 Gbps for 1080p, and 1.85 Gbps for 4k image resolutions at 30fps.

5.4.2 The Daisy Feature-extraction Block

The feature-extraction module is highly pipelined to perform stream processing of pixels. As mentioned above, the entire architecture comprises four processing steps that are heavily interleaved at the pixel, patch, and frame levels. This allows us to exploit the inherent parallelism in the application and perform computations with minimal delay. At a high level, the T-block is a single processing element that generates the T-block features sequentially. The patterns for spatial pooling in the S-block are stored in an on-chip memory along the borders of the 2D-array. The spatially pooled S-Block features are then produced at the output. The number of rows and columns in the G-Block array and the number of lanes in the S-Block array can be adjusted to achieve the desired energy and throughput scalability. Next, we provide more details on each block.

G-Block. Fig. 17 shows a block diagram of the implemented systolic-array architecture for 2d-convolution. Our architecture allows the inputs to be fed only once allowing maximum data reuse, which minimizes the bandwidth requirements from external memory. Further, the vector reduction process described above allows us to perform 2d convolution along any direction, with varying stride lengths, and kernel sizes.The systolic array is primarily used in the G-block.

T patches (of size $P \times P$ and centered at locations specified in the IPD output FIFO) are read out from external memory in block sizes of R pixels. In each iterations, these R pixels are processed in $R + 3C$ cycles to produce R processed 2d-convolution outputs. The processing core comprises a systolic array of 2d-processing elements (PEs), which are basically small multiply-accumulate (MAC) units and internal registers for fast-laning. As shown in Fig. 17, R input data vectors and the kernel elements stored in C columns are processed by the 2d-PEs sequentially. At any given point in time, the systolic array comprises fully- and partially-

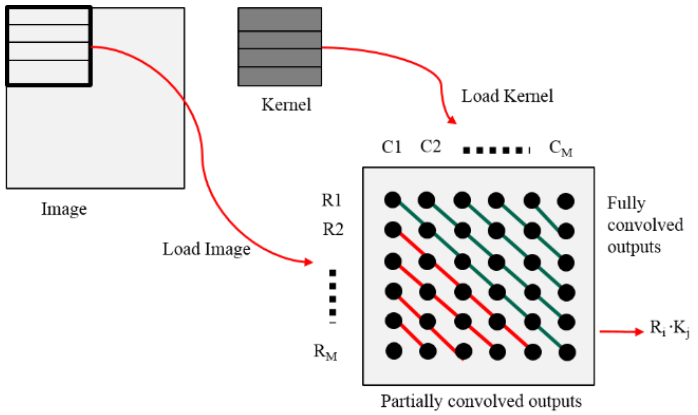


Fig. 18 At any given time, the systolic array comprises fully and partially convolved outputs that are reduced by the 1d PEs in the second level of processing.

convolved outputs. This aspect is shown in Fig. 18. As per the illustration, note in particular that the elements along the diagonal comprise the desired output that will be available after CM cycles. In order to accommodate the partially convolved outputs, we employ a set of 1d-PEs (accumulators) along the edge of the 2d-array.

The total memory requirements for the block are as follows: $RCd\times$ 8b for the I/O FIFOs of depth d (typically, 16) and $PC\times$ 24b to store the partially convolved outputs. If, pixels are re-fetched after IPD from external memory, the hardware requires an external memory bandwidth of $TP2\times$ 8b. However, in our implementation, we avoid going to external memory by adding local buffers between the IPD and feature-extraction blocks.

T, S, and N Blocks. Fig. 19 shows the block diagram of the T, S, and N blocks. The data path for the T-block comprises gradient-computation and quantization engines for the T1 (a), T1 (b), T2 (a), and T2 (b) modes of operation. In the S-block, we have a configurable number of parallel lanes for the spatial-pooling process. These lanes comprise comparators that read out Np pooling region boundaries from a local memory and compare with the current pixel locations. The output from the S-block is processed by the N-block, which comprises an efficient square-rooting algorithm and division module (based on CORDIC). The T-block outputs are buffered in a local memory of size $6(R+2)\times$ 24b and the pooling region boundaries are stored in a local SRAM memory of size $3Np\times$ 8b. The power consumption and performance of the S block can be adjusted by varying the number of lanes in the array. These are called the parallel S-block lanes and we study their impact ahead in the experimental results section (Sec. 6.2.3).

All data precisions are tuned to maximize the output signal-to-noise-ratio (SNR) for most images. The levels of parallelism in the system, the output precisions, memory sizes *etc.* can all be parameterized in the code. In conclusion, assuming no local data buffering between the IPD and daisy feature-extraction modules, the total mem-

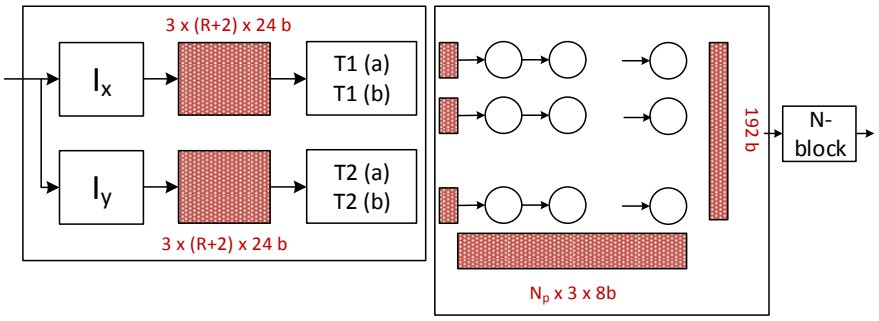


Fig. 19 Block diagram of the T, S, and N processing blocks. SAPPHIRE has low internal memory overheads: 207.38 kB for VGA, 257.32 kB for 1080p, and 331.11 kB for 4k image resolutions.

ory requirements of the feature-extraction block (for nominal ranges) are (assuming 64×64 patch size and 100 interest points): 1.2 kB (4×4 2d array and 25 pooling regions) for a frame resolution of VGA (128×128 patch size and 100 interest points) and 3.5 kB (8×8 2d array and 25 pooling regions) for a frame resolution of 720p HD. Since, in our implementation, we include local buffering between the IPD and feature-extraction modules, they work in a pipelined manner and thus the external data access bandwidth is completely masked. The total estimated storage capacity for IPD and feature-extraction is 207.38 kB for VGA, 257.32 kB for 1080p, and 331.11 kB for 4k image resolutions

5.4.3 The FV Feature-representation Block

The microarchitecture of the FV representation block is shown in Fig. 20. It comprises three processing elements, namely, Q-compute, FV-compute and Q-norm compute. We exploit parallelism across GMM clusters by ordering the Q and FV computations in an arrayed fashion. The GMM parameters (*i.e.*, μ, σ, and π) are stored in on-chip streaming memory elements. The daisy feature descriptors come in from the left, and are processed by the Q- and FV-compute elements. After one round of processing, the global feature memory is updated. This process is repeated across all GMM clusters – recall that the number of GMM clusters is an algorithmic parameter that is fixed during the initial design-space exploration phase. To maximize throughput, the GMM model parameters are shared across successive feature inputs in the Q and FV-compute elements. This sharing also saves us memory bandwidth. The power and performance of the FV block can be adjusted by varying the number of lanes in the processing element array. We revisit this aspect in Sec. 6.2.3.

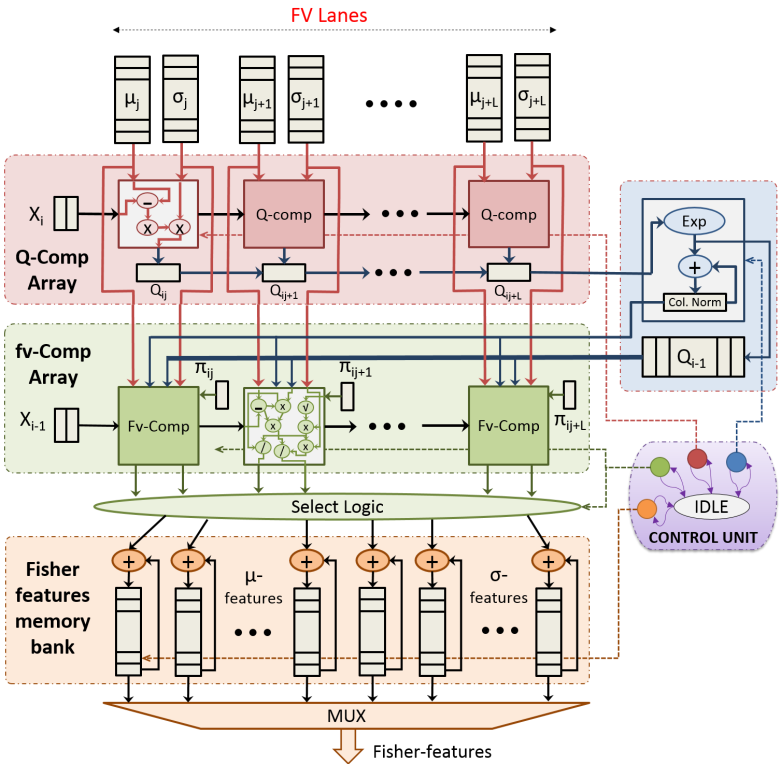


Fig. 20 Block diagram of the fisher-vector computation block. It involves three elements: Q computation, Q-norm computation, and FV computation. The GMM parameters are shared across Q and FV computations of successive patches. Figure reproduced from [25].

5.4.4 The SVM Feature-classification Block

Fig. 21 shows the microarchitecture of the SVM block. It comprises two types of PEs, namely, the dot-product unit (DPU) and the kernel-function unit (KFU). These units together realize the distance computation. Support vectors, which represent the trained model, are stored in a streaming memory bank along the borders of the DPU array. During on-line classification, the DPUs perform L1 vector reduction between the feature descriptors and the support vectors to compute the dot products. After this, the dot products are streamed out to the KFU, where the kernel function (representing the L2 reduction) and the distance score is computed. In our implementation, we only support linear and polynomial kernels, but provide easy extensibility options for other kernels. Finally, the distance score is used by the global decision unit (GDU). to compute the classifier output. Note that all of the previous operations are independent and can be parallelized. Note also that the execution time of the SVM is proportional to the number of DPU units (SVM lanes).

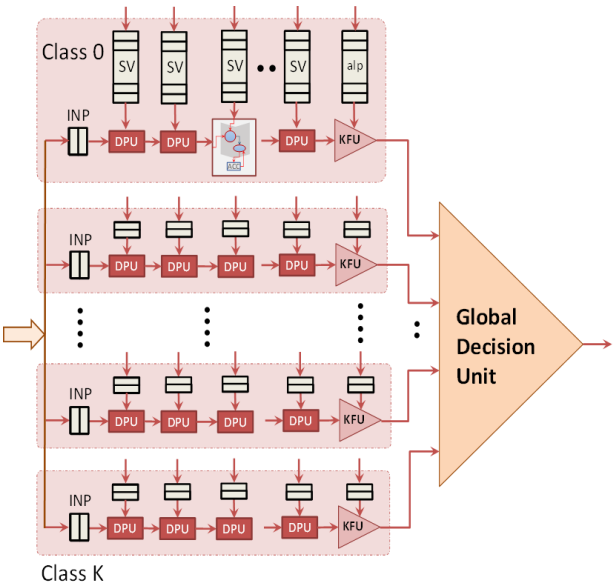


Fig. 21 Block diagram of the SVM classification block. Multiple (horizontal) processing lanes allow parallel processing of the FVs.

Through the various microarchitectural and hardware optimizations (*e.g.*, specialized processing elements, parallel stages, and multi-tiered pipelines) mentioned in this section, SAPPHIRE performs efficient image classification. The ability to scale performance and energy by adjusting the various design parameters is also a key attribute of the hardware architecture. We explore this aspect next.

6 SAPPHIRE Evaluation

We evaluate the performance and energy consumption of SAPPHIRE in an ASIC implementation. In this section, we describe about our experimental methodology. We then present results at various levels of the design hierarchy.

6.1 Experimental Methodology

In this section, we describe our methodology and the benchmarks that we used to evaluate the performance and energy consumption of SAPPHIRE.

Architecture-level evaluation. We implemented SAPPHIRE at the register-transfer logic (RTL) level using Verilog hardware description language (HDL). We synthesized it to an ASIC in a 45 nm SOI process using Synopsys Design Compiler. We

Table 2 Microarchitectural- and circuit-level parameters used in SAPPHIRE. Table reproduced from [25]

μ**Arch. params**	**Value**
G-Blk Rows/Cols	3/8
S-Blk Lanes	1
FV Lanes	2
SVM Lanes	4
Peak GOPS (Daisy,FV,SVM)	29 (18.5,6,2.5)

Circuit Params	**Value**
Feature size	45 nm SOI
Area	0.5 mm^2
Power (lkg+act)	51.8 mW
Gate Count	150k
Frequency	250 MHz

used Synopsys Power Compiler and Primetime to estimate the power consumption and delay of SAPPHIRE at the gate level, respectively. The microarchitectural- and circuit-level parameters that we used in our implementation are shown in Table 2. Since repeatedly simulating the algorithm at the gate-level was prohibitive in terms of runtime, we developed a cycle-accurate simulation model for the design. This model helped us estimate the hardware performance much more efficiently. For the estimations, we computed the energy consumption of SAPPHIRE as a product of the cycle count, operating frequency, and total power.

System-level energy modeling. We estimated the energy consumption in the end-to-end streaming system model (see Fig. 3) as follows:

$$E_{baseline} = E_{sense} + E_{compress} + E_{transmit} \tag{13}$$

where E_{sense}, $E_{compress}$, and $E_{transmit}$ are the energies for sensing, compression, and data transmission, respectively. We estimate the energy of the proposed system model (see Fig. 5) as follows:

$$E_{proposed} = E_{sense} + E_{SAPPHIRE} + (1-\gamma)\,(E_{compress} + E_{transmit}) \tag{14}$$

where γ is the defined as the fraction of the filtered frames (*i.e.*, $\gamma = (100 - FT)/100$, where FT is in percentage). To cover a broad spectrum of devices, we estimate each of these energies by assuming a slightly relaxed choice of components (when compared to Figs. 3 and 5). Specifically, we use the following numbers: a less aggressive low-power OmniVision VGA sensor (100.08 mW) [38], a light-weight MPEG encoder (20 mW and 5$\times$ compression) [39], and low-bandwidth 802.11a/g WiFi transmitter (45 nJ/bit at 20 Mbps) [40]. We also assumed a frame rate of 10 fps.

Application benchmarks. We used the four benchmarks mentioned earlier to evaluate the performance of SAPPHIRE. The first three (Caltech256, NORB, and PASCAL VOC) are static image benchmarks, while CamVid is a labeled video dataset. Across these benchmarks, we design SAPPHIRE to detect frames that contain one of 13 objects and filter the rest.

6.2 Experimental Results

In this section, we demonstrate the performance and energy savings at the system level due to SAPPHIRE. We also illustrate the impact of parameter tuning on the hardware energy.

6.2.1 System-level Energy Benefits Due to SAPPHIRE

Like we mentioned before, adding SAPPHIRE saves us communication energy at the cost of some extra computational energy. The energy required by SAPPHIRE is shown in comparison to the other components in Fig. 22. Observe that SAPPHIRE achieves a 1.4-3.0× (2.1× on average) improvement in system energy, while capturing over 90% of interesting frames in the datasets; recall that these numbers are what we used to estimate the battery recharge times in Fig. 5. At lower coverage levels, the energy benefits are much higher. For instance, they reach to about 3.6× and 5.1× on an average at 70-90% and 50-70% coverages, respectively. It is interesting to note that at lower coverage levels, the higher system-level energy savings come about even in the presence of additional communication energy costs. The figure also shows the energy overhead incurred due to SAPPHIRE as a fraction of the total system energy.

Fig. 23 shows the total energy costs of SAPPHIRE in comparison with the other system components for the different datasets. Compared to the baseline, we see that SAPPHIRE only contributes to about 6% of the overall system energy. This energy dis-proportionality between identifying interesting data *vs.* completely transmitting them is key to the applicability of SAPPHIRE. The energy contributions of SAPPHIRE increase to 28% at lower coverage levels since the overall system energy is also significantly lowered.

Fig. 24 shows how much energy savings can be achieved through SAPPHIRE at different FoI levels. Observe that the energy benefits provided by SAPPHIRE are bounded by the maximum number of frames that can be filtered out (*i.e.*, FoI). At higher values of FoI, the savings due to SAPPHIRE are lower. For instance, at ≥ 90% coverage, the savings reduce from 2.1 to 1.3× as FoI goes from 5 to 70%.

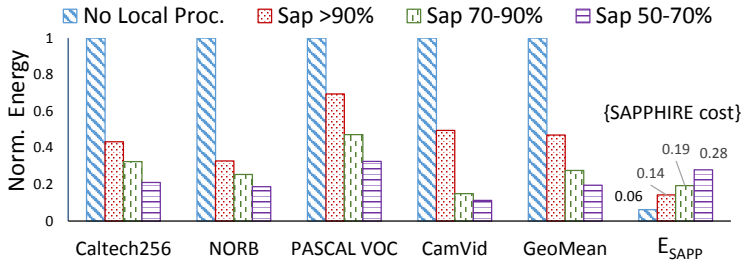


Fig. 22 SAPPHIRE costs 6% overhead but lowers system energy by 2.1×. This overhead increases to 28% at lower coverage levels, but the overall system energy is also reduced.

	Caltech256	NORB	Pascal	CamVid
Image Size	640 x 480 24 b/pix	96 x 96 24 b/pix	640 x 480 24 b/pix	960 x 720 24 b/pix
Bits/frame	1.5 Mbits	88 Kbits	1.5 Mbits	3.3 Mbits
E/frame Sense+	1.7 mJ	0.06 mJ	1.7 mJ	3.8 mJ
E/frame Compress$	8 mJ	0.27 mJ	8 mJ	8 mJ
E/frame Tx*	66 mJ	2 mJ	66 mJ	149 mJ

Fig. 23 Comparison between energy costs of SAPPHIRE and other system components.

However, as we observed in Fig. 4, in most context-aware applications, FoIs are low ($\leq$ 10%). Thus, most systems can benefit substantially by employing SAPPHIRE for local data filtering.

6.2.2 Runtime and Energy Breakdown of SAPPHIRE

Fig. 25, at the top, shows the percentage contributions to power and runtime, respectively, of the various computational elements in SAPPHIRE. Note that the sum of all runtimes does not equal 100% since the hardware is pipelined and more than one block may be concurrently active. For these results, we use the microarchitectural configuration of Table 2. At the bottom, the figure shows the breakdown in the normalized energy. Observe that the energy proportions for the various com-

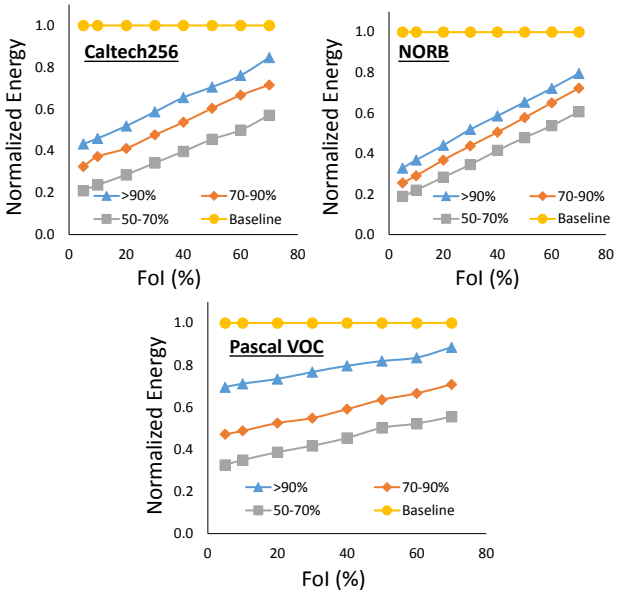


Fig. 24 SAPPHIRE saves more energy at lower FoI (typical of appl.). Figure adapted from [25].

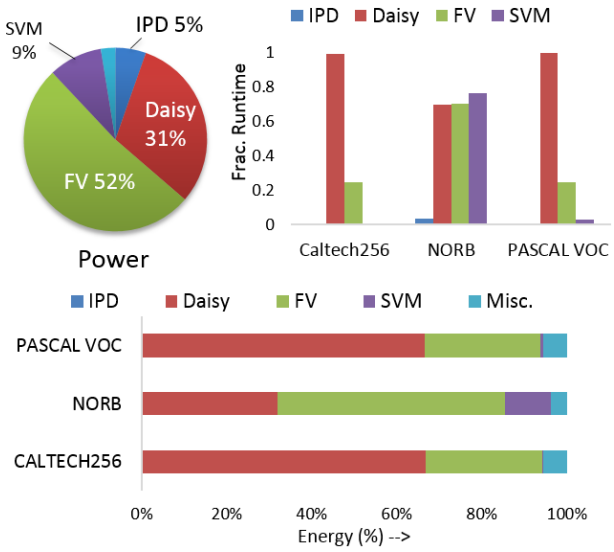


Fig. 25 Power, runtime, and energy breakdown of various computational blocks in SAPPHIRE. Figure adapted from [25].

putational elements depend on the complexity of the dataset. For instance, number of interest points are high in Caltech256, leading to a higher (∼90%) runtime for daisy feature extraction. This is in contrast with NORB, where the SVM classifier is active most of the time. Thus, we observe that the microarchitectural parameters of SAPPHIRE need to be tuned so that we can optimize the energy consumption for different datasets and applications. We explore this aspect next.

6.2.3 Microarchitectural Design-space Exploration

We perform an exhaustive search of the design space for the energy-optimal microarchitectural configuration of SAPPHIRE. Fig. 26(a) shows a scatter plot of performance [*i.e.*, achievable fps] *vs.* the normalized energy consumption per frame for various architectural configurations. In Fig. 26(b), the energy per frame is decoupled into two components, namely frame processing time (FPT) and power (the product of these is the energy/frame). The pareto optimal configurations that minimize the energy consumption are also shown in Fig. 26(a). The configurations are marked as a tuple comprising the number of parallel lanes in the G-, S-, FV- and SVM-blocks, and the operating frequency of SAPPHIRE. We see from the figure that the pareto-optimal configurations are not obtained by scaling just a single parameter, but a combination. Also, at lower FPS, the increase is FPT outweighs the corresponding decrease in power, thereby resulting in higher energy per frame. At higher FPS,however, the disproportional increase in power also leads to a higher frame energy. Thus the minimum energy configuration occurs at an FPS of ∼12 for

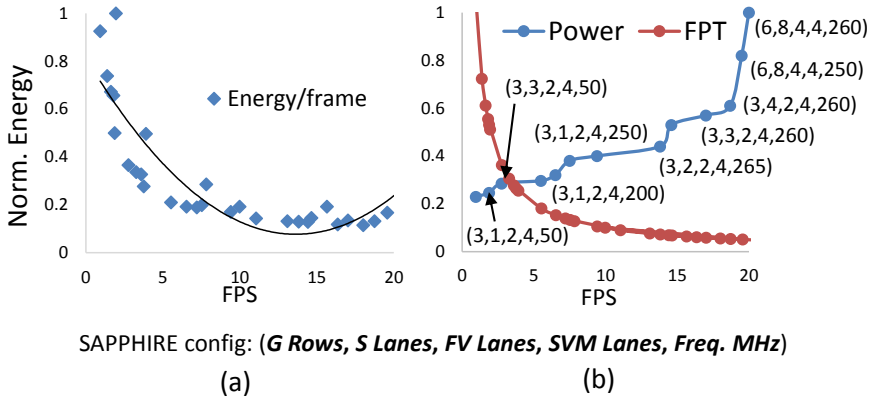


Fig. 26 Design space exploration showing the minimum-energy configuration of SAPPHIRE for Caltech256.

this dataset. Thus, SAPPHIRE allows us to achieve optimal energy configurations depending on the characteristics of the application data.

7 Conclusions

A range of emerging applications require portable devices to be continually ambient aware. However, this requires devices to be always on, leading to a large amount of sensed data. Transmitting this data to the cloud for analysis is power inefficient. In this article, we proposed the design of a hybrid system that employs local computations for image classification and the cloud for more complex processing. We chose a light-weight image-classification algorithm to keep the energy overheads low. We showed that even with this light-weight algorithm, we can achieve very high true positive rates at the cost of some extra false positives. This approach helped us filter out a substantial number of frames from video data in the device itself. In order to overcome the high processing latency in software, we also proposed a hardware-specialized accelerator called SAPPHIRE. This accelerator allowed us to perform image classification 235× faster than a CPU with a very low (3 mJ/frame) energy cost. Using multiple levels of pipelining and other architectural innovations, we were able to simultaneously achieve high performance and better energy efficiency in the end-to-end system. Thanks to the resulting communication energy reduction, we showed that our hybrid system using SAPPHIRE can bring down the overall system energy costs by 2.1×. Our system thus has the potential to prolong battery lives of many portable ambient-aware devices.

References

1. Baber C, Smith P, Cross J, Zasikowski D, and Hunter J (2005) Wearable technology for crime scene investigation. Proceedings of the IEEE International Symposium on Wearable Computers, 138–141.
2. Mann S (1998) WearCam (the wearable camera): Personal imaging systems for long-term use in wearable tetherless computer-mediated reality and personal photo/videographic memory prosthesis. Proceedings of the IEEE Int. Symposium on Wearable Computers, 124–131.
3. Kelly P, Marshall S J, Badland H, Kerr J, Oliver M, Doherty A R, and Foster C (2013) An ethical framework for automated, wearable cameras in health behavior research. American Journal of Preventive Medicine, 44(3):314–319, doi: 10.1016/j.amepre.2012.11.006.
4. D'Andrea R (2014) Can drones deliver? IEEE Transactions on Automation Science and Engineering, 138–141.
5. Navab N (2004) Developing killer apps for industrial augmented reality. IEEE Computer Graphics and Applications, 24(3):16–20.
6. Aleksya M, Rissanenb M J, Maczeya S, and Dixa M (2011) Wearable computing in industrial service applications. International Conference on Ambient Systems, Newtworking and Technologies, 5:394–400, doi: 10.1016/j.procs.2011.07.051.
7. Randell C (2005) Wearable computing: A review. Technical Report Number CSTR-06-004. University of Bristol.
8. Weinland D, Ronfard R, and Boyer E (2011) A survey of vision-based methods for action representation, segmentation and recognition. Elsevier Computer Vision and Image Understanding, 115(2):224–241, doi: 10.1016/j.cviu.2010.10.002.
9. Poppe R (2010) A survey on vision-based human action recognition. Elsevier Image and Vision Computing, 28(6):976–990, doi: 10.1016/j.imavis.2009.11.014.
10. Geronimo D, Lopez A M, Sappa A D, and Graf T (2009) Survey of pedestrian detection for advanced driver assistance systems. IEEE Transactions on Pattern Analysis and Machine Intelligence, 32(7):1239–1258, doi: 10.1109/TPAMI.2009.122.
11. Crevier D, and Lepage R (1997) Knowledge-based image understanding systems: A survey. Elsevier Computer Vision and Image Understanding, 67(2):161–185, doi: 10.1006/cviu.1996.0520.
12. LiKamWa R, Wang Z, Carroll A, Lin X F, and Zong L (2014) Draining our glass: An energy and heat characterization of Google Glass. Proceedings of Asia-Pacific Workshop on Systems, Article No. 10, doi: 10.1145/2637166.2637230.
13. Ha K, Chen Z, Hu W, Richter W, Pillai P, and Satyanarayanan M (2014) Towards wearable cognitive assistance. Proceedings of the International Conference on Mobile Systems, Applications, and Services, 68–81, doi: 10.1145/2594368.2594383.
14. Kemp R, Palmer N, Kielmann T, and Bal H (2012) Cuckoo: A computation offloading framework for smartphones. Mobile Computing, Applications, and Services. In Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecom. Engineering, 76:59–79.
15. Ra M-R, Sheth A, Mummert L, Pillai P, Wetherall D, and Govindan R (2011) Odessa: Enabling interactive perception applications on mobile devices. Proceedings of the International Conference on Mobile Systems, Applications, and Services, 43–56, doi: 10.1145/1999995.2000000.
16. Jia Z, Balasuriya A, and Challa S (2009) Vision based target tracking for autonomous land vehicle navigation: A brief survey. Recent Patents on Computer Science, 2(1):32–42.
17. Soro S and Heinzelman W (2009) A survey of visual sensor networks. Hindawi Advances in Multimedia, Article No. 640386, doi: 10.1155/2009/640386.
18. Kyono Y, Yonezawa T, Nozaki H, Keio M O, Keio T I, Keio J N, Takashio K, and Tokuda H (2013) EverCopter: Continuous and adaptive over-the-air sensing with detachable wired flying objects. Proceedings of the ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication, 299–302, doi: 10.1145/2494091.2494183.
19. Halperin D, Greenstein B, Sheth A, and Wetherall D (2010) Demystifying 802.11n power consumption. Proceedings of the International Conference on Power Aware Computing and Systems, Article No. 1.

20. Nishikawa T, Takahashi M, Hamada M, Takayanagi T, Arakida H, Machiada N, Yamamoto H, Fujiyoshi T, Matsumoto Y, Yamagishi O, Samata t, Asano A, Terazawa T, Ohmori K, Shirakura J, Watanabe Y, Nakamura H, Minami S, Kuroda T, and Furuyama T (2000) A 60MHz 240mW MPEG-4 video-phone LSI with 16Mb embedded DRAM. IEEE Journal of Solid-State Circuits, 35:1713–1721.
21. (2014) Worlds most power-efficient 1080p/60 high definition imag sensor for front-facing camera applications. OV2740 1080p Product Brief. Available online at www.ovt.com.
22. Brostow G, Shotton J, Fauquer J, and Cipolla R (2008) Segmentation and recognition using structure from motion point clouds. Proceedings of the European Conference on Computer Vision, 44–57, doi: 10.1007/978-3-540-88682-2_5.
23. Krizhevsky A, Sutskever I, and Hinton G E (2012) Imagenet classification with deep convolutional neural networks. Proceedings of Neural Information Processing Systems, 1106–1114.
24. Perronnin F, Sanchez J, and Mensink T (2010) Improving the fisher kernel for large-scale image classification. Proceedings of the European Conference on Computer Vision, 143–156.
25. Venkataramani S, Bahl V, Hua X-S, Liu J, Li J, Phillipose M, Priyantha B, and Shoaib M (2015) SAPPHIRE: An always-on context-aware computer-vision system for portable devices. Proceedings of Conference on Design Automation and Test in Europe, to appear.
26. Sanchez J, Perronnin F, Mensink T, and Jakob V (2013) Image classification with the Fisher Vector: Theory and practice. International Journal of Computer Vision, 105(3):222-245. doi: 10.1007/s11263-013-0636-x.
27. Winder S, Hua G, and Brown M (2009) Picking the best daisy. Proceedings of the International Conference on Computer Vision and Pattern Recognition.
28. LeCun Y, Huang F J, and Bottou L (2004) Learning methods for generic object recognition with invariance to pose and lighting. Proceedings of the International Conference on Computer Vision and Pattern Recognition. doi: 10.1109/CVPR.2004.1315150.
29. Everingham M, Ali E S M, Luc V G, Williams C K I, Winn J, and Zisserman A (2014) The pascal visual object classes challenge: A retrospective. International Journal of Computer Vision. 1–39, doi: 10.1007/s11263-014-0733-5.
30. Griffin G, Holum A, and Perona (2011) Caltech-256 object category dataset. Caltech Technical Report Number: CNS-TR-2007-001. Available online at: http://authors.library.caltech.edu/7694.
31. Harris C and Stephens M (1988) A combined corner and edge detector. Proceedings of the Fourth Alvey Vision Conference. 147–151.
32. Winder S A J and Brown M (2007) Learning local image descriptors. Proceedings of the International Conference on Computer Vision and Pattern Recognition. 1–8.
33. Winder S, Hua G, and Brown M (2009) Discriminative learning of local image descriptors. IEEE Transactions on Pattern Analysis and Machine Intelligence, 33(1): 43–57. doi: 10.1109/TPAMI.2010.54.
34. Shotton J, Johnson M, and Cipolla R (2008) Semantic texton forests for image categorization and segmentation. Proceedings of the International Conference on Computer Vision and Pattern Recognition, 1–8, doi: 10.1109/CVPR.2008.4587503.
35. He X, Yan S, Hu Y, Niyogi P, and Zhang H-J (2005) Face recognition using Laplacian faces. IEEE Transactions on Pattern Analysis and Machine Intelligence 27(3): 328 – 340.
36. Chen H-T, Chang H-W, and Liu T-L (2005) Local discriminant embedding and its variants. Proceedings of the International Conference on Computer Vision and Pattern Recognition, 846 – 853, doi: 10.1109/CVPR.2005.216.
37. Leijen D, Schulte W, and Burchardt S (2009) The design of a task parallel library. Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications, 227–242, doi: 10.1145/1640089.1640106.
38. (2014) OmniVision OV7735 Product Brief. Available online at www.ovt.com.
39. Chen S, Bermak A, and Wang Y (2011) A CMOS image sensor with on-chip image compression based on predictive boundary adaptation and memoryless QTD algorithm. IEEE Transactions on VLSI Systems, 19(4):538–547.
40. (2003) Low Power Advantage of 802.11a/g vs. 802.11b. Whitepaper, Texas Instruments. Available online at www.ti.com.

41. Jin J, Gokhale V, Dundar A, Krishnamurthy B, Martini B, and Culurciello E (2014) An efficient implementation of deep convolutional neural networks on a mobile coprocessor. IEEE International Midwest Symposium on Circuits and Systems, 133–136, doi: 10.1109/MWSCAS.2014.6908370.
42. Gokhale V, Jin J, Dundar A, Martini B, Culurciello E (2014) A 240 G-ops/s mobile coprocessor for deep neural networks. IEEE Conference on Computer Vision and Pattern Recognition Workshops, 696–701, doi: 10.1109/CVPRW.2014.106.
43. Chen T, Du Z, Sun N, Wang J, Wu C, Chen Y, and Temam O (2014) DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 269–284, doi: 10.1145/2541940.2541967.