SAPPHIRE: An Always-on Context-aware Computer Vision System for Portable Devices

Swagath Venkataramani
School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907
Email: venkata0@purdue.edu

Victor Bahl, Xian-Sheng Hua, Jie Liu, Jin Li,
Matthai Phillipose, Bodhi Priyantha, Mohammed Shoaib
Microsoft Research, One Microsoft Way, Redmond WA 98052
Email: {bahl,xshua,liuj,jinl,matthaip,bodhip,moshoaib}@microsoft.com

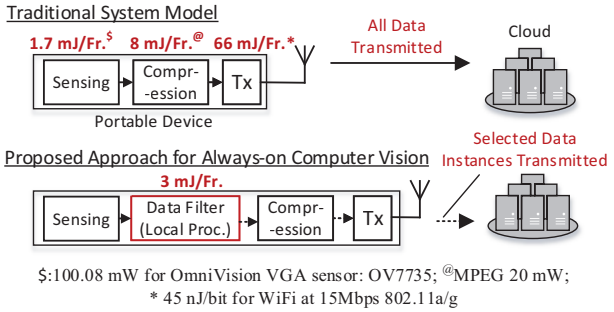
***Abstract*—Being aware of objects in the ambient provides a new dimension of context awareness. Towards this goal, we present a system that exploits powerful computer vision algorithms in the cloud by collecting data through always-on cameras on portable devices. To reduce comunication-energy costs, our system allows client devices to continually analyze streams of video and distill out frames that contain objects of interest. Through a dedicated image-classification engine SAPPHIRE, we show that if an object is found in 5% of all frames, we end up selecting 30% of them to be able to detect the object 90% of the time: 70% data reduction on the client device at a cost of ≤ 60mW of power (45nm ASIC). By doing so, we demonstrate system-level energy reductions of ≥ 2×. Thanks to multiple levels of pipelining and parallel vector-reduction stages, SAPPHIRE consumes only 3.0 mJ/frame and 38 pJ/OP – estimated to be lower by 11.4× than a 45 nm GPU – and a slightly higher level of peak performance (29 *vs.* 20 GFLOPS). Further, compared to a parallelized sofware implementation on a mobile CPU, it provides a processing speed up of up to 235× (1.81 s *vs.* 7.7 ms/frame), which is necessary to meet the real-time processing needs of an always-on context-aware system.**

***Keywords*—Always on, portable devices, computer vision, object recognition, hardware acceleration, energy efficiency**

I. Introduction

Emerging mobile applications require persistent context awareness [1]. One important form of awareness is about *what things exist in the vicinity of a device*. In this paper, we present the end-to-end design of a system that provides this form of context awareness. Devices can achieve ambient object-awareness by analyzing data from embedded sensors. The most promising of these is the camera. This is because, images and video are richer in light-field information as compared to other types of sensor data. Thus, keeping cameras always-on allows portable devices to be constantly object-aware, which can enable several interesting applications. For instance, if flying drones can detect obstacles in their path, they would be able to make use of robust navigation algorithms for efficient delivery. If dashboard or rooftop cameras in automobiles can detect pedestrians, traffic *etc.*, they would allow the use of complex decision algorithms, which are necessary for (semi) autonomous navigation.

System-level challenges. Although they provide rich data, always-on cameras on portable devices present several system-level design challenges [2]. Most of these arise due to the limited energy and computational resources available on portable devices. In the systems we consider, the eventual goal is to achieve *context awareness through image understanding*. This process thus requires the use of several computer vision algorithms. Among these, the foremost are the ones that are used for visual (object) recognition. In our work, we focus on this specific task. Recent results have shown that deep neural networks have the potential to provide state-of-the-art accuracy in visual recognition [3]. These algorithms employ dynamic decision models that require large memories, high-bandwidth data links, and compute capacities of up to several giga operations per second (OPS). With enormous potential parallelism, such algorithms form ideal workloads for acceleration in high-performance cloud clusters. Thus, they enable the system model shown at the top in Fig. 1 for visual object recognition. In this model, portable devices constantly stream camera data over a wireless link to the cloud. This approach, however, is undesirable due to the high energy cost of communication [4]. To enable real-time ambient object-awareness, there is thus a need to reduce the transmitted data from portable devices to the cloud.

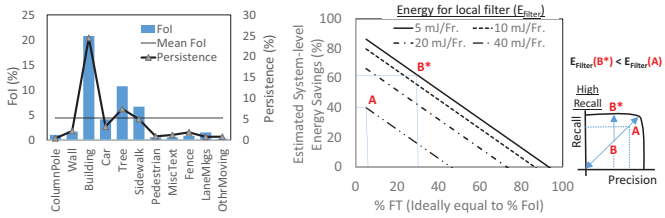


$:100.08 mW for OmniVision VGA sensor: OV7735; @MPEG 20 mW;
* 45 nJ/bit for WiFi at 15Mbps 802.11a/g

Fig. 1. Traditional system models are limited by communication energy. We propose to exploit light-weight local data filtering to overcome this limitation.

Our design approach. Data reduction on portable devices can be achieved by either lowering the frame rate of always-on cameras or by performing object recognition on the device itself. Both of these approaches have their limitations. On the one hand, naively reducing frame rates leads to a loss of information, while modest capabilities disallow the use of local object recognition on the other. In this paper, we propose an improved system model shown at the bottom in Fig. 1, which overcomes both of these limitations. We propose to perform low-complexity image classification on the device, while exploiting the power of the cloud for full-scale object recognition. Specifically, for the local computations, we employ a well-known algorithm for binary image classification, which allows substantial amounts of local data filtering with an energy cost that is well within the capacity of portable devices [5].

We motivate our approach through results from a well-known video-based object-recognition dataset shown in Fig. 2(a) [6]. The figure shows that frames-of-interest (FoI) (*i.e.*, those that contain relevant objects) typically only comprise a small percentage of all frames. Thus, if we can use an algorithm X to isolate those frames, we can achieve substantial energy savings at the system level as long as the algorithm's energy [$E_{filter}(X)$] is not very high. Fig. 2(b) illustrates two such potential algorithms A and B. When compared to B,



(a) (b)

Fig. 2. (a) For most objects, FoI ≤ 5% and they stay in view for at least 4.4% of contiguous frames. (b) When E_{filter} is low, FT ≥ FoI can give us substantial energy savings at the system level.

algorithm A achieves better precision and recall at the cost of higher processing energy. Suppose the FoI for a particular object is 5%. Ideally, we would want the frames transmitted (FT) to also equal 5%. Thanks to its high accuracy, suppose A is able to achieve this goal. We see from the figure that since E_{filter}(A) is high, the overall energy savings we get are not maximized. In fact, as E_{filter}(A) increases, the energy savings go down substantially. In this paper, we show how to bias a low-energy algorithm B so that it has high recall at the cost of a slightly lower precision. Thus, the resulting algorithm B* allows us to obtain higher energy savings than A despite having a higher FT percentage.

Given the short length of this paper, we restrict ourselves to a few key aspects of the system. The following are the major contributions we make:

- For the first time, we present a flexible system architecture that enables general purpose visual (object) recognition through always-on cameras on portable devices. We demonstrate total system energy reductions of 2-4× (depending on FoI) by exploiting powerful algorithms in the cloud along with locally running low-complexity classification algorithms on the device.
- We present an efficient approach to build configurable biased classifiers that achieve high (≥ 90%) recall and modest (≥ 50%) precision in image classification. Our system is thus able to achieve very high (≥ 92%) end-to-end recognition accuracy [3].
- We present a novel hierarchically-pipelined hardware architecture for image-classification that is able to operate at 10-360 fps (depending on frame size/data complexity), which is effectively 18-235× faster than a parallelized software implementation on a mobile CPU. We present synthesis results for the engine in a 45nm SOI process, which shows an average energy cost of 3.0 mJ/frame and 38 pJ/OP – lower by 11.4× than a 45 nm GPU – and comparable level of FLOPS.
- We present a new 2d-barrel convolution engine as a sub-component used in image-classification. This flexible engine performs 2-level vector reductions through a systolic array operation and can be configured for different stride lengths, directions, and kernel matrices. It also allows us to lower the memory bandwidth by performing maximum data reuse within the computation engine.

The rest of the paper is organized as follows. In Sec. II, we present related work. In Sec. III, we describe the classification algorithm along with results from a software implementation on a mobile CPU, which will motivate the need for the hardware specialization of the algorithm. In Sec. IV, we present the hardware architecture of SAPPHIRE, which is the low-energy image-classification engine that we propose. In Sec. V, we describe our experimental and evaluation methodology followed by results at the system, architecture, and circuit level in Sec. VI. Finally, we conclude in Sec. VII.

II. Related Work

Past research in this area has considered three distinct directions. The first addresses communication energy by exploiting compression [7]. The second optimizes the sensing energy by either designing low-energy image sensors [8], [9] or by tuning quality parameters such as frequency and sampling rates of existing sensors [10]. These set of directions attempt to trade image quality for energy efficiency. The final set of approaches avoid full-offloading of recognition to the cloud by performing partial computations on the device [11]. This last set is the most similar to our design model. In these systems, intermediate algorithmic variables (including features) are transmitted to the cloud, where the rest of the algorithm is run to completion. In contrast to these approaches, SAPPHIRE leverages the input data characteristics [low FoI, and high persistence (presence of the object in the field of view): see Fig. 2(a)], to completely eliminate continuous data transmission to the cloud. Data is transmitted only sporadically. We next describe details of the algorithm that enables SAPPHIRE to achieve this characteristic.

III. Algorithm for Low-energy Image Classification

When selecting an algorithm for image classification, it is important to note that our objective is to only use light-weight operations that can help isolate (but not recognize details of) potentially interesting frames. It is also useful if the classifier is programmable to detect any object of interest. We thus chose an algorithm that not only performed reasonably well in the ILSVRC competition, but also that which had a lower computational complexity [5]. The basic algorithm is illustrated in Fig. 3. It comprises four major computational blocks that we detail next.

Interest-point detection (IPD). This block helps identify pixels in a frame that potentially contain informative features such as edges, corner, blobs, ridges, *etc.* In our case, we utilize the Harris-Noble algorithm that detects corners in an image [12]. The algorithm iterates through every pixel in the image and considers pixels within a small neighborhood (*e.g.,* 3×3) to determine an attribute score, called the *corner measure*. This measure is obtained based on the computation of trace and determinant of a covariance matrix that is derived using the image gradients for all pixels in the window of interest. A pixel is deemed a corner, if its measure is largest among all abutting pixels and if it above a pre-specified threshold. This process is called non-maximum suppression. This process is computationally efficient and invariant to lighting, translation, and rotation.

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.*. Since we aim to have high-flexibility in the classifier, we chose the Daisy feature-extraction algorithm, which 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 [13]. As shown in Fig. 3, this algorithm comprises four computation sub-blocks:

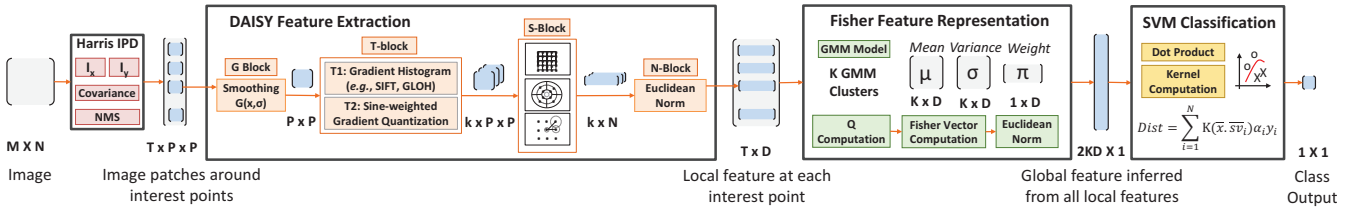


Fig. 3. Light-weight algorithm used for image-classification on the portable device.

- **G-Block:** The image patch is smoothed by convolving it with a 2d-Gaussian filter.
- **T-Block:** At each pixel, gradients along both horizontal and vertical directions are computed. The magnitude of the gradient vector is then apportioned into k (equals 4 in T1 and 8 in T2 mode) bins resulting in an output array of k feature maps, each of size P×P.
- **S-Block:** The feature maps from the T-block are then pooled along a grid that is foveated (1r-6/8s, 2r-6/8s), rectangular (rect), or polar. Each spatially pooled section produces one scalar value. Thus, if there are N points on the pooling grid, an N-dimensional vector is produced for each T-block feature map. When concatenated, these vectors thus lead to the final S-block feature output of dimensionality D (= kN).
- **N-Block:** The S-block features are normalized (non) iteratively using the l_2 norm to produce the final daisy feature vector of dimensionality D.

Feature representation. This block allows us to aggregate feature-vectors from all image patches to produce a vector of constant dimensionality. There are several algorithmic options for high-level feature representation including the bag-of-visual words, fisher vectors (FV), *etc.* [14]. We chose FV representation since it provides better classification performance, thanks to a richer Gaussian mixture model (GMM)-based representation of the visual vocabulary. The GMM is allowed to have K centroids each with a parameter set μ σ, and π corresponding to the mean, standard deviation, and proportion, respectively. The gradient of the log likelihood is computed with respect to the parameters of the model to represent every frame. The FV is the concatenation of these partial derivatives and describes in which direction the parameters of the model should be modified to best fit the data. This block thus produces a global feature vector of size 2KD.

SVM classification. A simple margin-based classifier [an support vector machine (SVM), in this case] is used to detect relevant frames based on a model that is learnt offline using training data. In SVMs, a set of vectors, called support vectors, determine the decision boundary. During online classification, the feature vector is used to compute a distance score that represents the probability with which the input belongs to a specific class. Modifying the decision boundary is thus key to biasing the classifier towards high recall, which is necessary for the reliable operation of SAPPHIRE.

A. Software implementation of the algorithm

We implemented a parallelized version of the algorithm in C# using task parallel library (TPL) provided by the .NET 4.5 framework. We evaluated the algorithm using four popular object-recognition datasets: Caltech256 [15], NORB [16], PASCAL VOC [17], and CamVid [6]. For these datasets, Fig. 4

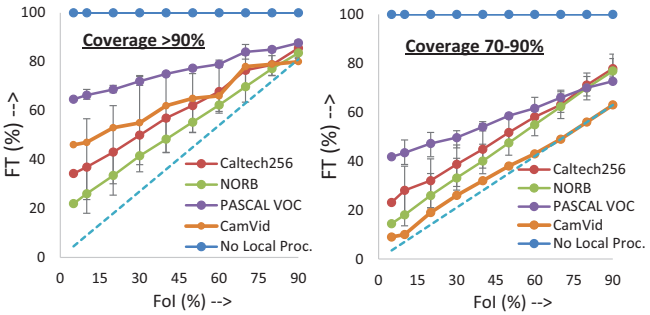


Fig. 4. FT, which is ≥ FoI at higher coverage values, begins to approach FoI as we relax the coverage constraints on the algorithm.

shows the trends in FT *vs.* FoI. The results are shown at two levels of *coverage*, which is a term we use synonymously with recall (the fraction of interesting frames that were detected). 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) at 90% and 70% coverage for the left and right sub-figures, respectively. We observe that without no on-device classification, FT is always 100%. Further, with local classification, we are able to filter out ~70% of the frames (averaged over all datasets) at FoI = 5%. This is quite substantial as it will directly translate into system level energy savings (as we show ahead in Sec. VI). We bias the SVM classifier to achieve different levels of coverage. This helps us explore the reduction in FT at lower levels of coverage. From the figure, we see that at 70-90% coverage, we are able to filter ~73% of the frames at 5% FoI.

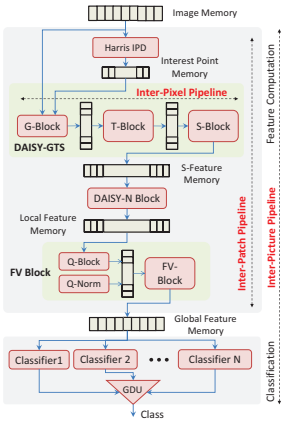
TABLE I. SOFTWARE IMPLEMENTATION OF IMAGE CLASSIFICATION INCURS A LARGE PROCESSING DELAY THAT IS UNACCEPTABLE FOR REAL-TIME CONTEXT-AWARE APPLICATIONS.

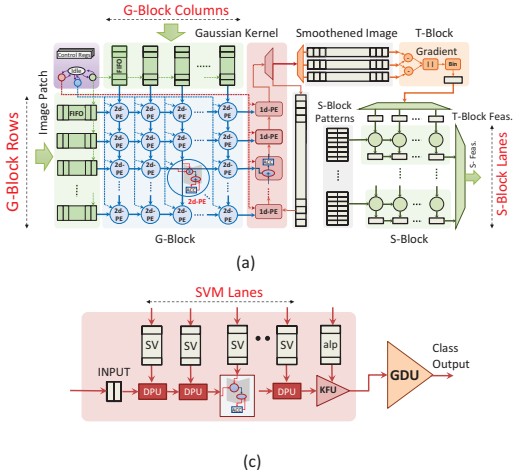
	Caltech256	NORB	PASCAL	CamVid
Image Size	640 × 480	96 × 96	640 × 480	720 × 960
MOPS	161	9	81	211
Time/frame (sec.)	3.5	0.33	1.6	4.5

Table I shows how the algorithm complexity varies depending on frame size, number of interest points, classifier model size, *etc.*. Across all datasets, we find that the mean complexity is quite low: ~50 MOPS. However, the software run-time on both a desktop (Core i7) and mobile CPU (Snapdragon 800) exceeds 1.8 sec./frame on average. This latency arises 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.

IV. SAPPHIRE: HARDWARE ARCHITECTURE

In this section, we propose a hardware-specialized engine for accelerating image classification on portable devices. We exploit computation patterns of the algorithm to achieve significant processing efficiency. A key feature of our microarchitecture is that it can be configured to obtain different power





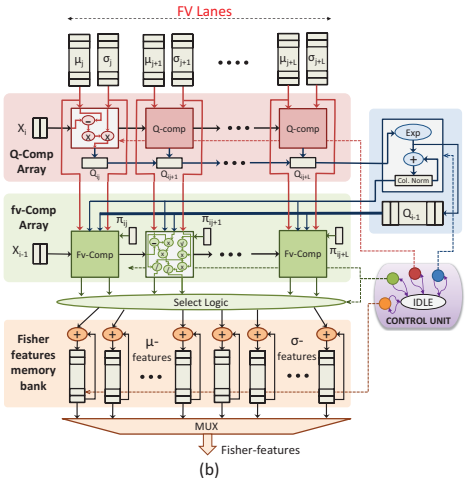


Fig. 5. Pipelining in SAPPHIRE. Fig. 6. Microarchitecture of (a) Daisy (b) FV and (c) SVM compute blocks.

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. We next outline a few key architectural innovations in SAPPHIRE.

A. Exploiting Data 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 exploit the data-level parallelism through pipelining. However, 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 Figure 5. A brief description of each level in the hierarchy is given below.

Inter-picture pipeline. The topmost tier is the inter-picture pipeline, which exploits pipelined parallelism across successive input frames. As illustrated in Fig. 5, it comprises two pipeline stages, namely feature computation and classification. Feature computation includes the IPD, Daisy and FV blocks, while classification block contains just the SVM. Thus, when global features of frame i are being computed, frame $i-1$ is concurrently processed by the classifier.

Inter-patch pipeline. This second tier of pipeline exploits parallelism within the feature computation stage of the inter-picture pipeline. In this case, image patches around different int erest points are processed concurrently in a pipeline, which comprises the following four stages: IPD, Daisy-GTS, Daisy-N, and FV. Interest points found by the IPD are pushed onto a first-in first-out (FIFO) memory and are utilized by Daisy-GTS to compute the S-block features. Daisy-N then normalizes this output to obtain local Daisy feature-vectors at that interest point. These vectors are consumed by the FV block, which iteratively updates the global feature memory. This process is repeated until the FIFO memory is emptied. Note that the stages of the inter-patch pipeline cannot be merged with the previous tier since global feature representation requires all feature vectors (interest points) of the frame to be evaluated. Due to this dependency, they must be independently operated as a second tier of pipeline.

Inter-pixel pipeline. This is the innermost tier of the hierarchy and is present within the DAISY-GTS block of the inter-patch pipeline. It leverages the parallelism across pixels in a patch by operating them in a pipeline. It contains 3 stages, namely the G-, T-, and S- Block. These together compute the S-Block feature output for each image patch in the frame.

To maximize throughput, it is imperative that we 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 inter-patch pipeline is proportional to the number of interest points, which varies across frames. Thus, in our implementation, we systematically optimize resource allocation for each block based on their criticality to the overall throughput.

B. Microarchitecture of Processing Blocks

In addition to pipelining, the algorithm also allows fine-grained parallel implementations within the various computation blocks of the pipeline. Many blocks involve a series of vector reduction operations, where in any vector $\vec{A}$ is reduced using a set of of vectors $B_1, \ldots B_m$ to produce an array of scalar outputs $s_1, \ldots s_m$. Each vector reduction involves an operation between individual elements of the vector, followed by accumulation $[s_i = \Sigma_k f(A_i, B_{jk})]$. An example of this pattern is the dot-product computation in 2-convolution (used in the G-block), where a series of multiply-accumulate operations are performed between an image patch and a convolution kernel [see Fig. 6(a)]. In our design, we employ arrays of specialized processing elements that are suitably interconnected to exploit this computation pattern. We next describe some microarchitectural details of the different blocks in SAPPHIRE.

1) Daisy Feature Computation Block: Daisy feature extraction comprises three components, namely G-, T-, and S-blocks, whose microarchitecture is shown in Fig. 6(a). For image smoothing in the G-block, we use a 2d-systolic array of vector-reduction processing elements that are connected to a 1d-array of accumulators located along the border. The vector reduction process described above allows us to perform 2d convolution along any direction, with varying stride lengths, and kernel sizes. The T-block is a single processing element that generates the T-block features sequentially. The patters for spatial pooling in the S-block are stored in an on-chip memory along the borders of the 2D-arrray. 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.

2) Fisher-vector Representation Block: The microarchitecture of the FV representation block is shown in Fig. 6(b).

It comprises three specialized processing elements,namely Q-compute, FV-compute and Q-norm compute. The Q and FV computation elements are arranged in an array allowing us to exploit parallelism across the GMM clusters. The μ, σ, and π vectors of the GMM are stored in (on-chip) streaming memory elements above the processing core. Feature vectors are fed 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. An important feature of this design is that the GMM model is shared across successive local feature inputs in the Q and FV-compute elements, thereby significantly reducing memory bandwidth. The power and performance of the FV representation block can be adjusted by varying the number of lanes in the processing element array.

3) SVM Classification Block: Fig. 6(c) shows the microarchitecture of the SVM block. It comprises two types of processing elements, namely the dot-product unit (DPU) and the kernel-function unit (KFU). The support vectors are stored in a streaming memory bank along the borders of the DPU array. In the classification mode, DPUs perform vector reduction to the global feature vector and the support vectors to compute the dot products. Next, the dot products are scanned out to the KFU, where the kernel function and the distance score is computed (in the current design, we support linear and polynomial kernels). Finally, the distance score is used by the global-decision unit (GDU) to compute the classifier output. The execution time of the SVM is directly proportional to the number of DPU units (SVM lanes).

Thus, SAPPHIRE performs efficient image classification using various specialized processing elements, parallel stages, and multi-tiered pipelines. The ability to scale performance and energy by adjusting the various microarchitectural parameters is also a key attribute of the hardware.

V. Experimental Methodology

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

Architecture-level evaluation. We implemented SAPPHIRE at the RTL level using Verilog HDL and synthesized it to a 45 nm SOI technology node using Synopsys Design Compiler. We used Synopsys Power Compiler and Primetime to estimate the power consumption and delay of our design at the gate level, respectively. The microarchitectural- and circuit-level parameters that we use in our implementation are shown in Table II. Since gate-level simulation of the entire algorithm took a prohibitively long runtime, we developed a cycle-accurate simulation model to help us estimate the hardware performance. We computed the energy consumption of SAPPHIRE as a product of the cycle count, operating frequency, and the total power.

TABLE II. Microarchitectural- and circuit-level parameters used in SAPPHIRE.

μArch. 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

System-level energy modeling. To help us calibrate against a baseline system, we consider the energy consumption of the system model shown at the top in Fig. 1. We estimate the baseline system-level energy as follows:

$$E_{baseline} = E_{sense} + E_{compress} + E_{transmit}$$

where E_{sense}, $E_{compress}$, and $E_{transmit}$ are the energies for sensing, compression, and data transmission, respectively. We estimate each of these energies by considering for following choice of components: a low-power OmniVision VGA sensor (100.08 mW) [9], MPEG encoder (20 mW and 5× compression) [8], and 802.11a/g WiFi transmitter (45 nJ/bit at 20 Mbps) [18]. We also assumed a frame rate of 10 fps. Further, we estimate the energy of the proposed system model (shown at the bottom in Fig. 1) as follows:

$$E_{proposed} = E_{sense} + E_{SAPPHIRE} + (1-\alpha)\left(E_{compress} + E_{transmit}\right)$$

where α is the defined as the fraction of the filtered frames (*i.e.*, $\alpha = (100 - FT)/100$, where FT is in percentage).

Application benchmarks. We evaluate the performance and energy consumption of SAPPHIRE using four object-recognition datasets. The first three (Caltech256, NORB, and PASCAL VOC) are static image benchmarks, while CamVid is a labeled video benchmark. Across these datasets, we design SAPPHIRE to detect frame that contain one of 13 objects and filter the rest.

GPU performance estimation. We follow a very simple approach to estimate the GPU performance and power numbers to a first order of accuracy. To keep the comparison fair, we consider three mobile GPUs (all in the 45 nm technology process): Apple A5 that run the PowerVR SGX 543MP2 from Imagination Technologies at 200 MHz, NVIDIA Tegra 3 that comprises a GeForce ULP GPU at 250 MHz, and the Snapdragon S4 that uses the Adreno 225 GPU at 300 MHz. For these GPUs, we use GFLOP values of 19.2, 18, and 19.2 and power values of 520, 650, and 600 mW, respectively. These numbers are obtained from the corresponding data sheets for the GPUs. Using the OPS computed from our cycle accurate simulation model, we estimate the power consumption and performance of the GPUs.

VI. Results

In this section, we demonstrate the performance and energy savings at the system level due to SAPPHIRE.

A. System-level Energy Benefits and Analysis

Fig. 7 shows the normalized energy consumption of SAPPHIRE compared to the baseline system. Results are shown at different coverage levels assuming 5% FoI (representative of many real-work always-on context-aware applications). The figure shows that SAPPHIRE achieves a 1.43× - 3.04× (2.12× on average) improvement in system energy, while capturing over 90% of interesting frames in the datasets. This reduction comes due to the filtering of a significant fraction of irrelevant frames on the client device, which saves us valuable communication energy. The benefits improve when the coverage requirements are relaxed – 3.61× and 5.12× on an average at 70-90% and 50-70% coverages, respectively. The figure also shows the energy overhead incurred due to SAPPHIRE as a fraction of the total system energy. Compared to the baseline, we see that SAPPHIRE only contributes to about 6% of the overall system energy. This energy disproportionality 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 decreased.

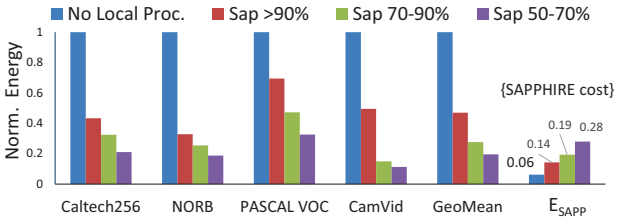


Fig. 7. SAPPHIRE costs 6% overhead but lowers system energy by 2.16×.

An interesting point to note is that the energy benefits provided by SAPPHIRE are bound by the maximum number of frames that can be filtered (*i.e.*, FoI). Fig. 8 shows that the system-level energy savings provided by SAPPHIRE decrease with increasing FoI: *e.g.* at ≥ 90% coverage, the savings reduce from 2.16× to 1.3× as FoI goes from 5% to 70%. However, in most context-aware applications, FoI is typically low (≤ 10%, see Fig. 2(a)) and thus always-on systems can gain substantial benefits from SAPPHIRE.

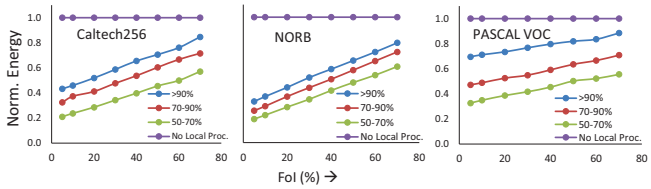


Fig. 8. SAPPHIRE saves more energy at lower FoI (typical of applications).

B. Runtime and Energy Breakdown

Based on the microarchitectural configuration of Table II, Fig. 9 shows the normalized energy as well as the percentage runtime and power consumption of the four compute blocks 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. Further, these proportions 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.

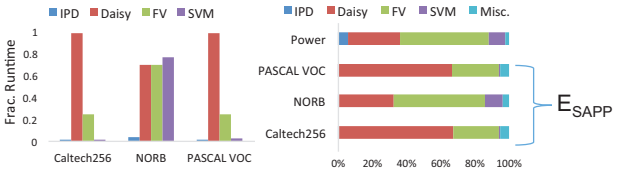


Fig. 9. Runtime and energy breakdown of compute blocks in SAPPHIRE.

VII. Conclusions

Building persistent context-aware systems requires us to aggregate data from always-on cameras on portable devices such as mobile phones, wearables, *etc.* The amount of data that these cameras generate costs communication overheads, which are unmanageable by most energy-constrained devices. In this paper, we proposed the design of a hybrid system that exploits local, on-device image classification as a means to filter data before transmission to the cloud, where advanced computer vision algorithms are employed to perform full-scale object recognition. Since, running image classification in software is slow, we developed a hardware-specialized accelerator called SAPPHIRE, which allowed us to perform classification 235× faster than a CPU with a very low (3 mJ/frame) energy cost – 11.4× lower than a GPU (see Fig. 10). We showed how to exploit multiple levels of pipeling and parallelism that is inherent in the algorithm to simultaneously achieve high performance and better energy efficiency. Using well-vetted image/video classification datasets, we showed that SAPPHIRE can bring down the overall system energy costs by 2.16×. We also observed that SAPPHIRE works well at low FoIs (typical in most applications) and has energy-optimal microarchitectural configurations that depend on the data at hand. Thus, thanks to SAPPHIRE, cameras on portable devices can remain always-on and enable a host of emerging context-aware applications that rely on the diverse advances in computer vision.

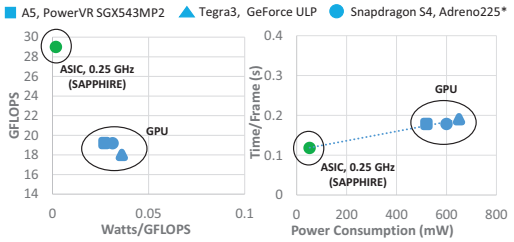


Fig. 10. We estimate that SAPPHIRE achieves a slightly higher performance (29 GFLOPS *vs.* 20 GFLOPS) and 11.4× better energy efficiency than a GPU.

References

[1] P. Bahl, M. Philipose, and L. Zhong. Vision: Cloud-powered sight for all: Showing the cloud what you see. In *Proc. Workshop Mobile Cloud Computing and Services*, pages 53–60, Jun. 2012.

[2] B. Ozer W. Wolf and Lv Tiehan. Smart cameras as embedded systems. *IEEE Computer*, 35(9):48–53, Sep. 2002.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proc. Neural Inform. Processing Syst.*, pages 1106–1114, Dec. 2012.

[4] M. Shoaib, J. Liu, and M. Phillipose. Energy scaling in multi-tiered sensing systems through compressive sensing. In *Proc. Custom Integrated Circuits Conference*, pages 1–8, Sep. 2014.

[5] F. Perronnin, J. Sanchez, and T. Mensink. Improving the fisher kernel for large-scale image classification. In *Proc. European Conf. Computer Vision*, pages 143–156, Sep. 2010.

[6] G. Brostow, Jamie Shotton, Julien Fauqueur, and Roberto Cipolla. Segmentation and recognition using structure from motion point clouds. In *Proc. European Conf. Comput. Vision*, pages 44–57, Oct. 2008.

[7] J.J. Ahmad, H. A. Khan, and S. A. Khayam. Energy efficient video compression for wireless sensor networks. In *Proc. Annu. Conf. Inform. Sci. and Syst.*, pages 629–634, Mar. 2009.

[8] S. Chen *et al.* A CMOS image sensor with on-chip image compression based on predictive boundary adaptation and memoryless QTD algorithm. *IEEE Trans. VLSI Syst.*, 19(4):538–547, Apr. 2011.

[9] OmniVision OV7735 Product Brief. Avail. online: http://www.ovt.com/.

[10] R. LiKamWa *et al.* Energy characterization and optimization of image sensing toward continuous mobile vision. In *Proc. Conf. Mobile Syst. Applicat. and Services*, pages 69–82, Jun. 2013.

[11] E. Cuervo *et al.* MAUI: Making smartphones last longer with code offload. In *Proc. Conf. Mobile Syst. Applicat. and Services*, pages 49–62, Jun. 2010.

[12] C. Harris and M. Stephens. A combined corner and edge detector. In *Proc. Alvey Vision Conf.*, pages 147–151, Sep. 1988.

[13] S. Winder *et al.* Picking the best daisy. In *Proc. Comput. Vision and Pattern Recognition*, pages 178–185, Jun. 2009.

[14] J. Sanchez *et al.* Image classification with the fisher vector: Theory and practice. *Int. J. Comput. Vision*, 105(3):222–245, 2013.

[15] G. Griffin *et al.* Caltech-256 object category dataset. Technical Report CNS-TR-2007-001, Jul. 2007.

[16] Y. LeCun *et al.* Learning methods for generic object recognition with invariance to pose and lighting. In *Proc. Comput. Vision and Pattern Recognition*, volume 2, pages 97–104, Jun. 2004.

[17] M. Everingham *et al.* The pascal visual object classes challenge: A retrospective. *Int. J. Comput. Vision*, pages 1–39, Jun. 2014.

[18] Texas Instruments Whitepaper. Low power advantage of 802.11a/g *vs.* 802.11b. Dec. 2003.