The Full Version Appears in the 45th International Symposium on Computer Architecture (ISCA 2018)

Algorithm-SoC Co-Design for Energy-Efficient Mobile Continuous Vision

Yuhao Zhu[1] Anand Samajdar[2] Matthew Mattina[3] Paul Whatmough[3]

[1]University of Rochester
[2]Georgia Institute of Technology
[3]Machine Learning & AI Research, ARM
yzhu@rochester.edu, anandsamajdar@gatech.edu
{matthew.mattina, paul.whatmough}@arm.com

Abstract

Continuous computer vision (CV) tasks increasingly rely on convolutional neural networks (CNN). However, CNNs have massive compute demands that far exceed the performance and energy constraints of mobile devices. In this paper, we propose an algorithm-architecture co-designed system, Euphrates, that simultaneously improves the energy-efficiency and performance of continuous vision tasks. Harnessing the insight that changes in pixel data between consecutive frames represents visual motion, we first propose an algorithm that leverages this motion information to relax the number of expensive CNN inferences required by continuous vision applications. We co-design a mobile System-on-a-Chip (SoC) architecture to maximize the efficiency of the new algorithm. The key to our architectural augmentation is to co-optimize different IP blocks in the vision pipeline collectively. Measurement and synthesis results show that Euphrates achieves up to 66% SoC-level energy savings, with only 1% accuracy loss. Our work demonstrates a promising first step toward tightly co-designing the mobile SoC architecture with vision algorithms.

1. Introduction

Computer vision (CV) is the cornerstone of many emerging application domains, such as advanced driver-assistance systems (ADAS) and augmented reality (AR). Traditionally, CV algorithms were dominated by hand-crafted features (e.g., Haar [29] and HOG [14]), coupled with a classifier such as a support vector machine (SVM) [13]. These algorithms have low complexity and are practical in constrained environments, but only achieve moderate accuracy. Recently, convolutional neural networks (CNNs) have rapidly displaced hand-crafted feature extraction, demonstrating significantly higher accuracy on a range of CV tasks including image classification [28], object detection [25], and visual tracking [24].

This paper focuses on *continuous vision* applications that extract high-level semantic information from real-time video streams. Continuous vision is challenging for mobile architects because it requires careful engineering of the mobile system as a whole. On one hand, continuous vision naturally encompasses multiple on-/off-chip hardware components such as the camera sensor, Image Signal Processor (ISP), DRAM, and various CV accelerators. Focusing only on one single System-on-a-Chip (SoC) component is unlikely to provide significant overall system-level improvement. On the other hand, unlike most mobile applications that have bursty behaviors [18], continuous vision applications are generally "always-on." As such, many familiar techniques that target bursty behaviors such as turbo boosting are not applicable. Thus, we must make wise SoC architecture decisions.

The compute requirement for continuous vision is enormous. Using object detection as an example, today's CNN accelerators could offer about 1 Tera-Operations-Per-Second (TOPS) peak compute capability under a typical 1 W mobile power budget [7, 11]. State-of-the-art, CNN-based approaches such as YOLOv2 [26], SSD [23], and Faster R-CNN [27] all have at least one order of magnitude higher compute requirements than accommodated in a mobile device. Reducing the CNN complexity (e.g., Tiny YOLO [25], which is a heavily truncated version of YOLO with 9/22 of its layers) or falling back to traditional hand-crafted features such as Haar [15] and HOG [30] lowers the compute demand, which, however, comes at a significant accuracy penalty.

The goal of our work is to improve the compute efficiency of continuous vision with small accuracy loss, thereby enabling new use cases on mobile devices. Our main contribution in this paper is an algorithm-SoC co-designed approach to achieve that goal. The key idea is to exploit the motion information inherent in real-time videos. Specifically, conventional continuous vision algorithms treat each frame as a standalone entity and thus execute an entire CNN inference on every frame. However, pixel changes across consecutive frames are not arbitrary; instead, they represent visual object motion. We propose a new algorithm that leverages the temporal pixel motion to extrapolate vision results with little computation while avoiding expensive CNN inferences on many frames.

We augment a conventional mobile SoC to support the new algorithm. Our SoC augmentations harness two architectural insights. First, we can greatly improve the compute efficiency while simplifying the architecture design by *exploiting the synergy between different SoC IP blocks*. Specifically, we observe that the pixel motion information is naturally generated by the ISP early in the vision pipeline owing to ISP's inherent algo-

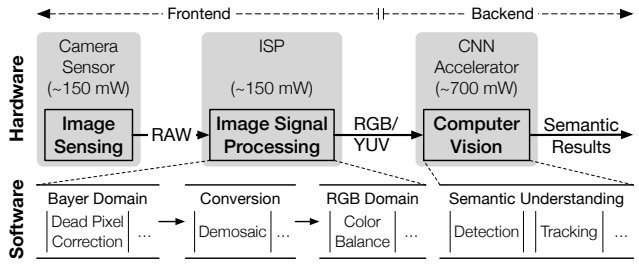
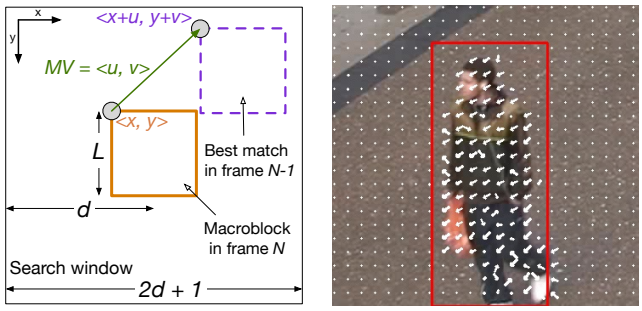


Fig. 1: A typical continuous computer vision pipeline.



(a) Block-matching. (b) Motion vectors.

Fig. 2: Motion estimation. (a): Block-matching example in a $(2d+1)\times(2d+1)$ search window. L is the macroblock size. (b): Each arrow reprensents an MB's motion vector. MBs in the foreground object have much more prounced motions than the background MBs.

rithms, and thus can be obtained with little compute overhead. We augment the SoC with a lightweight hardware extension that exposes the motion information to the rest of the SoC.

Second, although the new algorithm is light in compute, implementing it in software is energy-inefficient from a system perspective because it would frequently wake up the CPU, which is usually in the low-power mode during continuous vision tasks. Instead, we introduce the concept of a *motion controller*, which is a new hardware IP we propose that autonomously sequences the vision pipeline and performs motion extrapolation—all without interrupting the CPU. The motion controller's microarchitecture resembles a simple microcontroller, and thus incurs very low design and area cost.

We develop Euphrates, a proof-of-concept system of our algorithm-SoC co-designed approach. We evaluate Euphrates on object detection, which is critical to many continuous vision scenarios such as ADAS and AR. Based on real hardware measurements and RTL implementations, we show that Euphrates doubles the object detection rate while reducing the SoC energy by 66% at the cost of less than 1% accuracy loss.

2. Background and Motivation

We first give an overview of the continuous vision pipeline from the software and hardware perspectives (Sec. 2.1). In particular, we highlight an important design trend in the vision frontend where ISPs are increasingly incorporating motion estimation, which we exploit in this paper (Sec. 2.2). Finally, we briefly describe the block-based motion estimation algorithm and its data structures that are used in this paper (Sec. 2.3).

2.1. The Mobile Continuous Vision Pipeline

The continuous vision pipeline consists of two parts: a frontend and a backend, as shown in Fig. 1. The frontend prepares pixel data for the backend, which in turn extracts semantic information for high-level decision making.

The frontend uses (off-chip) camera sensors to capture light and produce RAW pixels that are transmitted to the mobile SoC, typically over the MIPI camera serial interface (CSI) [6]. Once on-chip, the Image Signal Processor (ISP) transforms the RAW data in the Bayer domain to pixels in the RGB/YUV domain through a series of imaging algorithms. In architecture terms, the ISP is a specialized IP block based on a pipeline of mostly stencil operations operating on a set of local SRAMs (i.e., line-buffers). The vision frontend typically stores frames in the main memory for communicating with the vision backend due to the large size of the image data.

The continuous vision backend extracts useful information from images through semantic-level tasks such as object detection. Traditionally, these algorithms are spread across DSP, GPU, and CPU. Recently, mobile SoC vendors have deployed dedicated CNN accelerators such as the Neural Engine in the iPhoneX [2] and the CNN co-processor in the HPU [8].

2.2. Motion Estimation in ISPs

ISPs are integrating sophisticated computational photography algorithms that are traditionally performed as separate image enhancement tasks, possibly off-line, using CPUs or GPUs. Among new algorithms that ISPs are integrating is motion estimation, which estimates how pixels move between consecutive frames. Motion estimation is at the center of many imaging algorithms such as temporal denoising, video stabilization (i.e., anti-shake), and frame upsampling. For instance, a temporal denoising algorithm [20, 22] uses pixel motion information to replace noisy pixels with their noise-free counterparts in the previous frame. Motion-based imaging algorithms are traditionally performed in GPUs or CPUs later in the vision pipeline, but they are increasingly subsumed into ISPs to improve compute efficiency. Commercial examples of motion-enabled camera ISPs include ARM Mali C-71 [3] and Hikvision video surveillance cameras [4], just to name a few based on public information.

2.3. Motion Estimation using Block Matching

Among various motion estimation algorithms, block-matching (BM) [19] is widely used in ISP algorithms such as temporal denoising [20]. The key idea of BM is to divide a frame into multiple $L\times L$ macroblocks (MB), and search in the previous frame for the closest match for each MB using Sum of Absolute Differences (SAD) of all L^2 pixels as the matching metric. The search is performed within a 2-D search window with $(2d+1)$ pixels in both vertical and horizontal directions,

where d is the search range. Fig. 2a illustrates the concepts.

Eventually, BM calculates a *motion vector* (MV) for each MB, which represents the location offset between the MB and its closest match in the previous frame as illustrated in Fig. 2a. Critically, this offset can be used as an estimation of the MB's motion. For instance, an MV $<u,v>$ for an MB at location $<x,y>$ indicates that the MB is moved from location $<x+u,y+v>$ in the previous frame. Fig. 2b visualizes the motion vectors in a frame. Note that MVs can be encoded efficiently. An MV requires $\lceil log_2(2d+1) \rceil$ bits for each direction, which equates to just 1 byte of storage under a typical d of seven.

3. Motion-based Continuous Vision Algorithm

This section first provides an overview of the motion-based algorithm (Sec. 3.1). We then discuss two important design decisions made for the algorithm (Sec. 3.2 and Sec. 3.3).

3.1. Overview

Euphrates makes a distinction between two frame types: Inference frame (I-frame) and Extrapolation frame (E-frame). An I-frame refers to a frame where vision computation such as detection and tracking is executed using expensive CNN inference with the frame pixel data as input. In contrast, an E-frame refers to a frame where visual results are generated by extrapolating ROIs from the previous frame, which itself could either be an I-frame or an E-frame.

Intuitively, increasing the ratio of E-frames to I-frames reduces the number of costly CNN inferences, thereby enabling higher frame rates while improving energy-efficiency. However, this strategy must have little accuracy impact to be useful. As such, the challenge of our algorithm is to strike a balance between accuracy and efficiency. We identify two aspects that affect the accuracy-efficiency trade-off: *how* to extrapolate from previous frame, and *when* to perform extrapolation.

3.2. How to Extrapolate

The goal of extrapolation is to estimate the ROI(s) for the current frame without CNN inference by using the motion vectors generated by the ISP. Our hypothesis is that the average motion of all pixels in a visual field can largely estimate the field's global motion. Thus, the first step in the algorithm calculates the average motion vector (μ) for a given ROI according to Equ. 1, where N denotes the total number of pixels bounded by the ROI, and $\overrightarrow{v_i}$ denotes the motion vector of the i^{th} bounded pixel. It is important to note that the ISP generates MVs at a macroblock-granularity, and as such each pixel inherits the MV from the MB it belongs to.

$$\mu = \sum_i^N \overrightarrow{v_i} \; / \; N \tag{1}$$

$$\alpha_F^i = 1 - \frac{SAD_F^i}{255 \times L^2} \tag{2}$$

$$MV_F = \beta \cdot \mu_F + (1-\beta) \cdot MV_{F-1} \tag{3}$$

Extrapolating purely based on average motion, however, is vulnerable to motion vector noise due to the nature of the block-based motion estimation algorithm. For instance, when a visual object is occluded or blurred, the block-matching algorithm may not find a good match within the search window.

To represent how noisy an MV is, we associate each MV with a confidence value. We empirically find that an MV's confidence is highly correlated with its SAD value, which is generated during block-matching. Intuitively, a higher SAD value indicates a lower confidence, and vice versa. Equ. 2 formulates the confidence calculation, where SAD_F^i denotes the SAD value of the i^{th} macroblock in frame F, and L denotes the dimension of the macroblocks. Effectively, we normalize an MV's SAD to the maximum possible value (i.e., $255 \times L^2$) and regulate the resultant confidence (α_F^i) to fall between $[0,1]$. We then derive the confidence for an ROI by averaging the confidences of all the MVs encapsulated by the ROI.

Given the confidence value, the idea to filter noisy motions is to assign a high weight to the average MV in the current frame (μ_F) if its confidence is high. Otherwise, we put more weight on the motion from previous frame. Essentially, this is equivalent to apply a high-pass filter that passes high confidence MVs. In a recursive fashion, this filter can be achieved by using Equ. 3, where MV_F denotes the final motion vector for frame F, MV_{F-1} denotes the motion vector for the previous frame, and β is the filter coefficient that is determined by α. In our experiments, we find it effective to use a simple piece-wise function that sets β to α if α is greater than a threshold and to 0.5 otherwise. In the end, we linearly apply the final motion vector (MV_F) to an ROI's location in the previous frame to update its new location. That is: $R_F = R_{F-1} + MV_F$

3.3. When to Extrapolate

Another important aspect of the extrapolation algorithm is to decide which frames to execute CNN inference on, and which frames to extrapolate. To simplify the discussion, we introduce the notion of Extrapolation Window (EW), which is the number of consecutive frames between two I-frames (exclusive). Intuitively, as the EW increases, the compute efficiency improves, but errors introduced by extrapolation also start accumulating, and vice versa. Therefore, EW is an important knob that determines the trade-off between compute efficiency and accuracy. Euphrates provides two modes regarding EW control: a constant mode and an adaptive mode.

In the constant mode, EW is statically set as a constant. A useful feature of the constant mode is that it offers predictable performance and energy-efficiency improvements. For instance, under an EW of two one can roughly estimate that the amount of computation per frame is reduced by half, translating to 2× performance increase or 50% energy savings.

However, the constant mode can not adapt to extrapolation inaccuracies. Without loss of generality, we introduce a dynamic control mechanism to respond to inaccuracies introduced by motion extrapolation. Specifically, whenever a

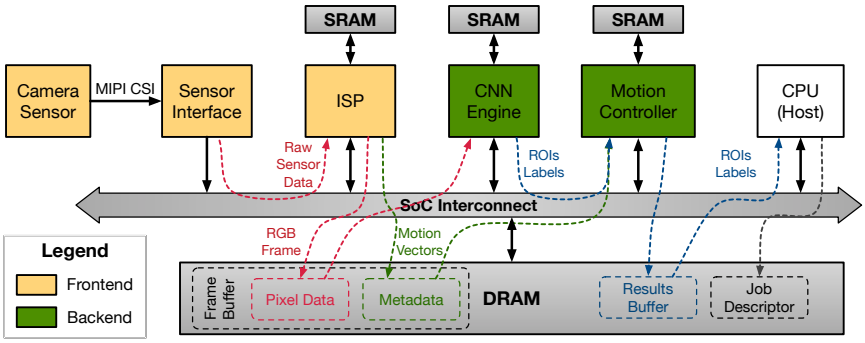


Fig. 3: Block diagram of the augmented continuous vision subsystem in a mobile SoC.

CNN inference is triggered we compare its results with the ones obtained from extrapolation. If the difference is larger than a particular threshold, we incrementally shrink the EW; conversely, if the difference is consistently lower than the threshold, we incrementally increase the EW.

4. Architecture Support

This section starts from a state-of-the-art mobile SoC and shows how to co-design the SoC architecture with the proposed algorithm. After providing an overview (Sec. 4.1), we describe the hardware augmentations required in the frontend (Sec. 4.2) and backend of the vision subsystem (Sec. 4.3).

4.1. System Overview

Fig. 3 illustrates the augmented mobile SoC architecture. In particular, we propose two architectural extensions. First, motivated by the synergy between the various motion-enabled imaging algorithms in the ISP and our motion extrapolation CV algorithm, we augment the ISP to expose the motion vectors to the vision backend. Second, to coordinate the backend under the new algorithm without significant CPU intervention, we propose a new hardware IP called the motion controller. The frontend and backend communicate through the system interconnect and DRAM.

Our proposed system works in the following way. The CPU initially configures the IPs in the vision pipeline, and initiates a vision task by writing a job descriptor. The camera sensor module captures real-time raw images, which are fed into the ISP. The ISP generates, for each frame, both pixel data and metadata that are transferred to an allocated frame buffer in DRAM. The motion vectors and the corresponding confidence data are packed as part of the metadata in the frame buffer.

The motion controller sequences operations in the backend and coordinates with the CNN engine. It directs the CNN engine to read image pixel data to execute an inference pass for each I-frame. The inference results, such as predicted ROIs and possibly classification labels for detected objects, are written to dedicated memory mapped registers in the motion controller through the system interconnect. The motion controller combines the CNN inference data and the motion vector data to extrapolate the results for E-frames.

4.2. Augmenting the Vision Frontend

To simplify discussion, this paper assumes that the motion vectors are generated by the temporal denoising (TD) stage in an ISP. The motion estimation block in the TD stage calculates the MVs and uses a small local SRAM to buffer them, which are then used by the motion compensation block to denoise the current frame. After the current frame is temporally-denoised the SRAM space for its MVs can be recycled.

We propose to expose the MVs by storing them in the metadata section of the frame buffer, which resides in the DRAM and is accessed by other SoC IPs through the system MMU. This augmentation is implemented by modifying the ISP's sequencer to properly configure the DMA engine.

Piggybacking the existing frame buffer mechanism rather than adding a dedicated link between the ISP and the vision backend has the minimum design cost with negligible memory traffic overhead. Specifically, a 1080p frame (1920×1080) with a 16×16 macroblock size will produce 8,100 motion vectors, equivalent to only about 8 KB per frame (Recall from Sec. 2.3 that each motion vector can be encoded in one byte), which is a very small fraction of the 6 MB frame pixel data that is already committed to the frame buffer.

4.3. Augmenting the Vision Backend

We augment the vision backend with a new IP called motion controller. Its job is two-fold. First, it executes the motion extrapolation algorithm. Second, it coordinates with the CNN engine without interrupting the CPU. The CNN accelerator is left intact and we reuse its interfaces to the SoC interconnect.

We design the motion controller engine as a micro-controller (μC) like IP, similar to many sensor co-processors such as Apple's Motion Co-processor [1]. It sits on the system interconnect, alongside the CNN accelerator. Fig. 4 shows the microarchitecture of the extrapolation controller. Important data and control flows in the figure are numbered. The motion controller is assigned the master role and the CNN engine acts as a slave in the system. The master IP controls the slave IP by using its sequencer to program the slave's memory-mapped registers (❶ and ❷ in Fig. 4). The slave IP always returns the computation results to the master IP (❸) instead of directly interacting with the CPU. We choose this master-slave sepa-

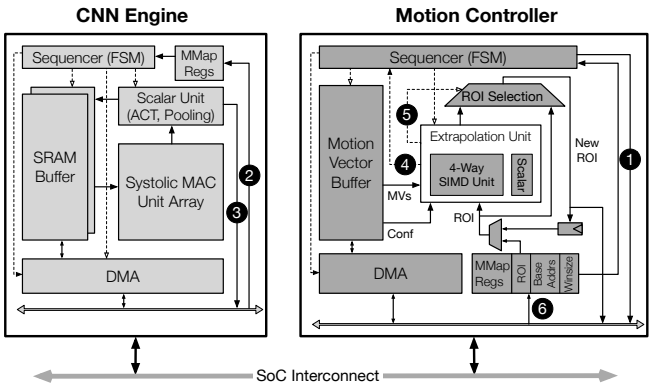


Fig. 4: Euphrates adds the motion controller to the vision backend, alongside an existing, unmodified CNN inference accelerator. Dash lines are control signals and solid lines represent the data flow.

ration, instead of the other way around, because it allows us to implement all the control logics such as adaptive EW completely in the extrapolator engine without making assumptions about the CNN accelerator's internals.

The core of the motion controller's datapath is an extrapolation unit which includes a SIMD unit and a scalar unit. The extrapolation operation is highly parallel (Sec. 3.2), making SIMD a nature fit. The scalar unit is primarily responsible for generating two signals: one that controls the EW size in the adaptive mode (❹) and the other that chooses between inferenced and extrapolated results (❺). The IP also has a set of memory-mapped registers that are programmed by the CPU initially and receive CNN engine's inference results (❻).

5. Implementation and Experimental Setup

This section introduces our hardware modeling methodology (Sec. 5.1) and software infrastructure (Sec. 5.2).

5.1. Hardware Setup

Our evaluation methodology is based on the GemDroid [12] SoC simulator. We develop a functional model, a performance model, and an energy model for evaluating the continuous vision pipeline. The functional model takes in video streams to mimic real-time camera capture and implements the extrapolation algorithm in OpenCV, from which we derive accuracy results. We integrate models for the camera sensor, the ISP, the CNN accelerator, and the motion controller. The performance model simulates the timing of each IP and cross-IP activities, from which we tabulate SoC events that are fed into the power model for energy estimation.

Whenever possible, we calibrate the power model by measuring the Nvidia Jetson TX2 module [5], which is widely used in mobile vision systems. We develop RTL models for the NN accelerator and the motion controller, and refer to public data sheets when direct measurement is unavailable. Overall, the power consumption introduced by the motion controller is only 2.2 mW. The area is also negligible (35,000 um^2). Table 1 shows the details about our modeled SoC.

Table 1: Details about the modeled vision SoC.

Component	Specification
Camera Sensor	ON Semi AR1335, 1080p @ 60 FPS
ISP	768 MHz, 1080p @ 60 FPS
NN Accelerator (NNX)	24×24 systolic MAC array 1.5 MB double-buffered local SRAM 3-channel, 128bit AXI4 DMA Engine
Motion Controller (MC)	4-wide SIMD datapath 8KB local SRAM buffer 3-channel, 128bit AXI4 DMA Engine
DRAM	4-channel LPDDR3, 25.6 GB/s peak BW

5.2. Software Setup

We evaluate Euphrates under one popular mobile continuous vision scenarios: object detection. In particular, we study a state-of-the-art object detection CNN called YOLOv2 [25, 26], which achieves the best accuracy and performance among all the object detectors. YOLOv2 requires over 3.4 TOPS compute capability at 60 FPS, significantly exceeding the mobile compute budget. For comparison purposes, we also evaluate a scaled-down version of YOLOv2 called Tiny YOLO. At the cost of 20% accuracy degradation [10], Tiny YOLO reduces the compute requirement to 675 GOPS, which is within the capability of our CNN accelerator.

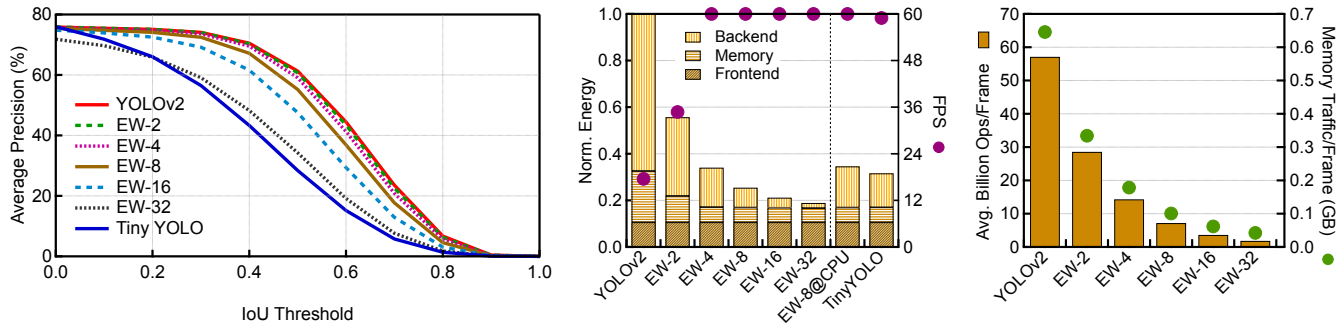
We evaluate object detection using an in-house video dataset. We could not use public object detection benchmarks (e.g., Pascal VOC 2007 [9]) as they are mostly composed of standalone images without temporal correlation as in real-time video streams. We use the standard Intersect-over-Union (IoU) score as the accuracy metric for object detection [17, 21].

6. Evaluation

Euphrates doubles the achieved FPS with 45% energy saving at the cost of only 0.58% accuracy loss. Compared to the conventional approach of reducing the CNN compute cost by scaling down the network size, Euphrates achieves a higher frame rate, lower energy consumption, and higher accuracy.

Accuracy Results Fig. 5a compares the average precision (AP) between baseline YOLOv2 and Euphrates under different extrapolation window sizes (EW-N, where N ranges from 2 to 32 in stride of powers of 2). For a comprehensive comparison, we vary the IoU ratio from 0 (no overlap) to 1 (perfect overlap). Each $<x, y>$ point corresponds to the percentage of detections (y) that are above a given IoU ratio (x). Overall, the AP declines as the IoU ratio increases.

Replacing expensive NN inference with cheap motion extrapolation has negligible accuracy loss. EW-2 and EW-4 both achieve a success rate close to the baseline YOLOv2, represented by the close proximity of their corresponding curves in Fig. 5a. Specifically, under an IoU of 0.5, which is commonly regarded as an acceptable detection threshold [16], EW-2 loses only 0.58% accuracy compared to the baseline YOLOv2.



(a) Average precision comparison.

(b) Energy and FPS comparison.

(c) Compute and memory comparison.

Fig. 5: Average precision, normalized energy consumption, and FPS comparisons between various object detection schemes. Energy is broken-down into three main components: backend (CNN engine and motion controller), main memory, and frontend (sensor and ISP).

Energy and Performance The energy savings and FPS improvements are significant. Fig. 5b shows the energy consumptions of different mechanisms normalized to the baseline YOLOv2. We also overlay the FPS results on the right *y*-axis. The energy consumption is split into three parts: frontend (sensor and ISP), main memory, and backend (CNN engine and motion controller). The vision frontend is configured to produce frames at a constant 60 FPS in this experiment. Thus, the frontend energy is the same across different schemes.

The baseline YOLOv2 consumes the highest energy and can only achieve about 17 FPS, which is far from real-time. As we increase EW, the total energy consumption drops and the FPS improves. Specifically, EW-2 reduces the total energy consumption by 45% and improves the frame rate from 17 to 35; EW-4 reduces the energy by 66% and achieves real-time frame rate at 60 FPS. The frame rate caps at EW-4, limited by the frontend. Extrapolating beyond eight consecutive frames have higher accuracy loss with only marginal energy improvements.

The significant energy efficiency and performance improvements come from two sources: relaxing the compute in the backend and reducing the SoC memory traffic. Fig. 5c shows the amount of arithmetic operations and SoC-level memory traffic (both reads and writes) per frame under various Euphrates settings. As EW increases, more expensive CNN inferences are replaced with cheap extrapolations (Sec. 3.2), resulting in significant energy savings. Euphrates also reduces the amount of SoC memory traffic.This is because E-frames access only the motion vector data, and thus avoid the huge memory traffic induced by executing the CNNs (SRAM spills).

Finally, the second to last column in Fig. 5b shows the total energy of EW-8 when extrapolation is performed on CPU. EW-8 with CPU-based extrapolation consumes almost as high energy as EW-4, essentially negating the benefits of extrapolation. This confirms that our architecture choice of using a dedicated motion controller IP to achieve task autonomy is important to realizing the full benefits in the vision pipeline.

Tiny YOLO Comparison One common way of reducing energy consumption and improving FPS is to reduce the CNN model complexity. For instance, Tiny YOLO uses only nine of YOLOv2's 24 convolutional layers, and thus has an 80% MAC operations reduction.

However, we find that exploiting the temporal motion information is a more effective approach to improve object detection efficiency than simply truncating a complex network. The bottom curve in Fig. 5a shows the average precision of Tiny YOLO. Although Tiny YOLO executes 20% of YOLOv2's MAC operations, its accuracy is even lower than EW-32, whose computation requirement is only 3.2% of YOLOv2. Meanwhile, Tiny YOLO consumes about 1.5 $\times$ energy at a lower FPS compared to EW-32 as shown in Fig. 5b.

7. Conclusion

Delivering real-time continuous vision in an energy-efficient manner is a tall order for mobile system design. To overcome the energy-efficiency barrier, we must expand the research horizon from individual accelerators toward holistically co-designing different mobile SoC components. This paper demonstrates one promising approach that leverages the temporal motion information naturally produced by the vision frontend (i.e., imaging) to reduce the compute demand of the vision backend (i.e., object detection and tracking). Future developments should look beyond motion data and expand the scope to other on/off-chip components. We hope our work serves the first step in a promising new direction of research.

References

[1] "Apple Motion Coprocessors." https://en.wikipedia.org/wiki/Apple_motion_coprocessors 4

[2] "Apple's Neural Engine Infuses the iPhone with AI Smarts." https://www.wired.com/story/apples-neural-engine-infuses-the-iphone-with-ai-smarts/ 2

[3] "ARM Mali Camera." https://www.arm.com/products/graphics-and-multimedia/mali-camera 2

[4] "Hikvision Advanced Image Processing: Noise Reduction." http://oversea-download.hikvision.com/UploadFile/file/Hikvision_Advanced_Image_Processing--Noise_Reduction.pdf 2

[5] "Jetson TX2 Module." http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html 5

[6] "MIPI Camera Serial Interface 2 (MIPI CSI-2)." https://www.mipi.org/specifications/csi-2 2

[7] "Movidius Myriad X VPU Product Brief." https://uploads.movidius.com/1503874473-MyriadXVPU_ProductBriefaug25.pdf 1

[8] "Second Version of HoloLens HPU will Incorporate AI Coprocessor for Implementing DNNs." https://www.microsoft.com/en-us/research/blog/second-version-hololens-hpu-will-incorporate-ai-coprocessor-implementing-dnns/ 2

[9] "The PASCAL Visual Object Classes Challenge 2007." http://host.robots.ox.ac.uk/pascal/VOC/voc2007/ 5

[10] "YOLO: Real-Time Object Detection." https://pjreddie.com/darknet/yolo/ 5

[11] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan, and V. Toma, "Always-on Vision Processing Unit for Mobile Applications," *IEEE Micro*, 2015. 1

[12] N. Chidambaram Nachiappan, P. Yedlapalli, N. Soundararajan, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "GemDroid: A Framework to Evaluate Mobile Platforms," 2014. 5

[13] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge university press, 2000. 1

[14] N. Dalal and B. Triggs, "Histograms of Oriented Gradients for Human Detection," in *Proc. of CVPR*, 2005. 1

[15] P. Dollár, R. Appel, S. Belongie, and P. Perona, "Fast Feature Pyramids for Object Detection," *PAMI*, 2014. 1

[16] M. Everingham, S. M. A. Eslami, L. V. Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge: A Retrospective," *IJCV*, 2015. 5

[17] M. Everingham, L. V. Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes (VOC) Challenge," *IJCV*, 2009. 5

[18] M. Halpern, Y. Zhu, and V. J. Reddi, "Mobile CPU's Rise to Power: Quantifying the Impact of Generational Mobile CPU Design Trends on Performance, Energy, and User Satisfaction," in *Proc. of HPCA*, 2016. 1

[19] M. Jakubowski and G. Pastuszak, "Block-based Motion Estimation Algorithms–A Survey," *Opto-Electronics Review*, 2013. 2

[20] H. Ji, C. Liu, Z. Shen, and Y. Xu, "Robust Video Denoising using Low Rank Matrix Completion," in *Proc. of CVPR*, 2010. 2

[21] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common Objects in Context," in *Proc. of ECCV*, 2014. 5

[22] C. Liu and W. T. Freeman, "A High-Quality Video Denoising Algorithm based on Reliable Motion Estimation," in *Proc. of ECCV*, 2010. 2

[23] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single Shot Multibox Detector," in *Proc. of ECCV*, 2016. 1

[24] H. Nam and B. Han, "Learning Multi-Domain Convolutional Neural Networks for Visual Tracking," in *Proc. of CVPR*, 2016. 1

[25] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *Proc. of CVPR*, 2016. 1, 5

[26] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," in *arXiv:1612.08242*, 2016. 1, 5

[27] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-time Object Detection with Region Proposal Networks," in *Proc. of NIPS*, 2015. 1

[28] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *Proc. of ICLR*, 2014. 1

[29] P. Viola and M. J. Jones, "Robust Real-time Object Detection," *IJCV*, 2004. 1

[30] J. Yan, Z. Lei, L. Wen, and S. Z. Li, "The Fastest Deformable Part Model for Object Detection," in *Proc. of CVPR*, 2014. 1